USENIX

Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation *Cambridge, MA, USA, April 11–13, 2007*

conference

proceedings

4th USENIX Symposium on Networked Systems Design & Implementation

Cambridge, MA, USA

April 11–13, 2007

Sponsored by

The USENIX Association

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

in cooperation with ACM SIGCOMM & ACM SIGOPS

For additional copies of these proceedings contact:

USENIX Association, 2560 Ninth Street, Suite 215, Berkeley, CA 94710 USA
Phone: 510-528-8649 • FAX: 510-548-5738 • office@usenix.org • http://www.usenix.org

The price is $40 for members and $50 for nonmembers.
Outside the U.S.A. and Canada, please add
$20 per copy for postage (via air printed matter).

Thanks to Our Sponsors

Google™ intel®

cisco™ hp® invent

Microsoft® Research vmware®

Thanks to Our Media Sponsors

ACM *Queue*	*Linux Journal*
GRIDtoday	*Linux Pro Magazine*
HPCwire	SNIA
ITtoolbox	*Sys Admin*

ISBN 1-931971-51-X

USENIX Association

Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI ’07)

April 11–13, 2007
Cambridge, MA, USA

Symposium Organizers

Program Co-Chairs

Hari Balakrishnan, *Massachusetts Institute of Technology*
Peter Druschel, *Max Planck Institute for Software Systems*

Program Committee

Jon Crowcroft, *Cambridge University*
Mike Dahlin, *University of Texas at Austin*
Deepak Ganesan, *University of Massachusetts Amherst*
Albert Greenberg, *AT&T Labs—Research*
Krishna Gummadi, *Max Planck Institute for Software Systems*
Rebecca Isaacs, *Microsoft Research, Cambridge*
Brad Karp, *University College, London*
Anne-Marie Kermarrec, *INRIA/IRISA*
Jinyang Li, *New York University*
Barbara Liskov, *Massachusetts Institute of Technology*
Petros Maniatis, *Intel Research, Berkeley*
Eugene Ng, *Rice University*
Sylvia Ratnasamy, *Intel Research, Berkeley*
Srinivasan Seshan, *Carnegie Mellon University*
Anees Shaikh, *IBM T.J. Watson Research Center*
Emin Gün Sirer, *Cornell University*
Alex Snoeren, *University of California, San Diego*
Chandu Thekkath, *Microsoft Research, Silicon Valley*

Poster Session Chairs

Krishna Gummadi, *Max Planck Institute for Software Systems*
Jinyang Li, *New York University*

Steering Committee

Thomas Anderson, *University of Washington*
Mike Jones, *Microsoft Research*
Greg Minshall
Robert Morris, *Massachusetts Institute of Technology*
Mike Schroeder, *Microsoft Research*
Amin Vahdat, *University of California, San Diego*
Ellie Young, *USENIX*

The USENIX Association Staff

External Reviewers

Aditya Akella
Periklis Akritidis
Gautam Altekar
Yair Amir
David Andersen
Paul Barham
Ranjita Bhagwan
Bobby Bhattacharjee
Nikita Borisov
Jeff Chase
Guillaume Chelius
Byung-Gon Chun
Manuel Costa
David Cottingham
Frank Dabek
Boris Dragovic
Adam Dunkels
Patrick Eugster
Nick Feamster
Rodrigo Fonseca
Paul Francis
Michael Freedman
Anh-Tuan Gai
Tal Garfinkel
Ben Greenstein
Steve Gribble
Rachid Guerraoui
Ramakrishna Gummadi
Andreas Haeberlen
Theo Hong
Wenjun Hu
Hai Huang
Arun Iyengar
John Jannotti
Glenn Judd
Evangelia Kalyvianaki
Dina Katabi
Idit Keidar
Eddie Kohler
Ramakrishna Kotla
Christian Kreibich
Kevin Lai
Ed Lee
Philip Levis
Harry Li
Menghow Lim
Boon Loo
John MacCormick
Sam Madden
Ratul Mahajan
Dave Maltz
Z. Morley Mao
John Miller
Andrew Moore
Edson Moreira
Robert Morris
Erich Nahum
Dushyanth Narayanan
Rolf Neugebauer
Christoph Neumann
James Newsome
Jason Nieh
Vivek Pai
Dina Papagiannaki
Andrea Passarella
Prashant Pradhan
Patrick Reynolds
Sean Rhea
Tom Rodeheffer
Rodrigo Rodrigues
Timothy Roscoe
Mema Roussopoulos
Romit Roy Choudhury
Yaoping Ruan
Sambit Sahu
Asad Samar
Stefan Saroiu
James Scott
Marc Shapiro
Atul Singh
Emil Sit
Ion Stoica
Lakshminarayanan Subramanian
Shu Tao
Dimitris Terzis
Renu Tewari
Kobus van der Merwe
Arun Venkataramani
Michael Walfish
Andrew Warfield
Janet Wiener
Ted Wobber
Junfeng Yang
Jian Yin
Haifeng Yu
Yuan Yu
Ming Zhang

4th USENIX Symposium on Networked Systems Design & Implementation (NSDI '07)
April 11–13, 2007
Cambridge, MA, USA

Thursday, April 12

Friday, April 13

Index of Authors

Message from the Program Co-Chairs

We are excited to bring these Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07) to you. There is no doubt now that NSDI has established itself as a top-tier conference and as the leading venue for innovative papers on the boundary between networking and computer systems. The authors who have submitted papers to NSDI over the past few years deserve full credit for this rapid success: they have submitted and continue to submit excellent papers, many of which describe and evaluate working systems.

This year offers a strong and interesting program across a wide range of areas in networked systems. The program also includes a poster session that features work in progress and a keynote by Ron Rivest on electronic voting systems.

We received 113 submissions this year, of which the program committee selected 27 for inclusion in these proceedings and for presentation at the conference. The committee reviewed the submissions in two rounds. In the first round, all submissions received a total of at least three reviews, including at least two reviews from a PC member. Based on these reviews, we selected about half of the submissions for further consideration in the second round. Each of these papers received at least two more reviews, including at least one more review from a PC member. A total of 523 reviews were written by the program committee and external reviewers. These reviews and extensive deliberations helped the program committee make its selection at a meeting in Cambridge, MA, in December 2006. Each accepted paper was then shepherded by a PC member.

We are extremely grateful to the members of the program committee for their dedication and hard work. Faced with a heavy reviewing load within a relatively short period of time, they diligently delivered detailed, high-quality, and thoughtful reviews. The many external reviewers who provided their expertise and time also deserve a special vote of thanks. Finally, as always, we thank Ellie Young, Jane-Ellen Long, Anne Dickison, Casey Henderson, and the rest of the USENIX staff, who are a pleasure to work with and who somehow take care of all the organizational magic.

Hari Balakrishnan, ***Massachusetts Institute of Technology***
Peter Druschel, ***Max Planck Institute for Software Systems***
Program Co-Chairs

Do incentives build robustness in BitTorrent?

Michael Piatek* Tomas Isdal* Thomas Anderson* Arvind Krishnamurthy* Arun Venkataramani†

Abstract

A fundamental problem with many peer-to-peer systems is the tendency for users to "free ride"—to consume resources without contributing to the system. The popular file distribution tool BitTorrent was explicitly designed to address this problem, using a tit-for-tat reciprocity strategy to provide positive incentives for nodes to contribute resources to the swarm. While BitTorrent has been extremely successful, we show that its incentive mechanism is not robust to strategic clients. Through performance modeling parameterized by real world traces, we demonstrate that all peers contribute resources that do not directly improve their performance. We use these results to drive the design and implementation of *BitTyrant*, a strategic BitTorrent client that provides a median 70% performance gain for a 1 Mbit client on live Internet swarms. We further show that when applied universally, strategic clients can hurt average per-swarm performance compared to today's BitTorrent client implementations.

1 Introduction

A fundamental problem with many peer-to-peer systems is the tendency of users to "free ride"—consume resources without contributing to the system. In early peer-to-peer systems such as Napster, the novelty factor sufficed to draw plentiful participation from peers. Subsequent peer-to-peer systems recognized and attempted to address the free riding problem; however, their fixes proved to be unsatisfactory, e.g., "incentive priorities" in Kazaa could be spoofed; currency in MojoNation was cumbersome; and the AudioGalaxy Satellite model of "always-on" clients has not been taken up. More recently, BitTorrent, a popular file distribution tool based on a *swarming* protocol, proposed a tit-for-tat (TFT) strategy aimed at incenting peers to contribute resources to the system and discouraging free riders.

The tremendous success of BitTorrent suggests that TFT is successful at inducing contributions from rational peers. Moreover, the bilateral nature of TFT allows for enforcement without a centralized trusted infrastructure. The consensus appears to be that "incentives build robustness in BitTorrent" [3, 17, 2, 11].

In this paper, we question this widely held belief. To this end, we first conduct a large measurement study of real BitTorrent swarms to understand the diversity of BitTorrent clients in use today, realistic distributions of peer upload capacities, and possible avenues of strategic peer behavior in popular clients. Based on these measurements, we develop a simple model of BitTorrent to correlate upload and download rates of peers. We parametrize this model with the measured distribution of peer upload capacities and discover the presence of significant *altruism* in BitTorrent, i.e., all peers regularly make contributions to the system that do not directly improve their performance. Intrigued by this observation, we revisit the following question: *can a strategic peer game BitTorrent to significantly improve its download performance for the same level of upload contribution?*

Our primary contribution is to settle this question in the affirmative. Based on the insights gained from our model, we design and implement *BitTyrant*, a modified BitTorrent client designed to benefit strategic peers. The key idea is to carefully select peers and contribution rates so as to maximize download per unit of upload bandwidth. The strategic behavior of *BitTyrant* is executed simply through policy modifications to existing clients without any change to the BitTorrent protocol. We evaluate *BitTyrant* performance on real swarms, establishing that all peers, regardless of upload capacity, can significantly improve download performance while reducing upload contributions. For example, a client with 1 Mb/s upload capacity receives a median 70% performance gain from using *BitTyrant*.

How does use of *BitTyrant* by many peers in a swarm affect performance? We find that peers individually benefit from *BitTyrant*'s strategic behavior, irrespective of whether or not other peers are using *BitTyrant*. Peers not using *BitTyrant* can experience degraded performance due to the absence of altruisitic contributions. Taken together, these results suggest that "incentives do not build robustness in BitTorrent".

Robustness requires that performance does not degrade if peers attempt to strategically manipulate the system, a condition BitTorrent does not meet today. Although BitTorrent peers ostensibly make contributions to improve performance, we show that much of this contribution is unnecessary and can be reallocated or withheld while still improving performance for strategic users. Average download times currently depend on significant altruism from high capacity peers that, when withheld, reduces performance for all users.

In addition to our primary contribution, *BitTyrant*, our

*Dept. of Computer Science and Engineering, Univ. of Washington
†Dept. of Computer Science, Univ. of Massachusetts Amherst

efforts to measure and model altruism in BitTorrent are independently noteworthy. First, although modeling BitTorrent has seen a large body of work (see Section 6), our model is simpler and still suffices to capture the correlation between upload and download rates for real swarms. Second, existing studies recognizing altruism in BitTorrent consider small simulated settings or few swarms that poorly capture the diversity of deployed BitTorrent clients, peer capacities, churn, and network conditions. Our evaluation is more comprehensive. We use trace driven modeling to drive the design of *BitTyrant*, which we then evaluate on more than 100 popular, real world swarms as well as synthetic swarms on PlanetLab. Finally, we make *BitTyrant* available publicly as well as source code and anonymized traces gathered in our large-scale measurement study.

The remainder of this paper is organized as follows. Section 2 provides an overview of the BitTorrent protocol and our measurement data, which we use to parametrize our model. Section 3 develops a simple model illustrating the sources and extent of altruism in BitTorrent. Section 4 presents *BitTyrant*, a modified BitTorrent client for strategic peers, which we evaluate in Section 5. In Section 6, we discuss related work and conclude in Section 7.

2 BitTorrent overview

This section presents an overview of the BitTorrent protocol, its implementation parameters, and the measurement data we use to seed our model.

2.1 Protocol

BitTorrent focuses on bulk data transfer. All users in a particular swarm are interested in obtaining the same file or set of files. In order to initially connect to a swarm, peers download a metadata file, called a *torrent*, from a content provider, usually via a normal HTTP request. This metadata specifies the name and size of the file to be downloaded, as well as SHA-1 fingerprints of the data blocks (typically 64–512 KB) that comprise the content to be downloaded. These fingerprints are used to verify data integrity. The metadata file also specifies the address of a *tracker* server for the torrent, which coordinates interactions between peers participating in the swarm. Peers contact the tracker upon startup and departure as well as periodically as the download progresses, usually with a frequency of 15 minutes. The tracker maintains a list of currently active peers and delivers a random subset of these to clients, upon request.

Users in possession of the complete file, called *seeds*, redistribute small blocks to other participants in the swarm. Peers exchange blocks and control information with a set of directly connected peers we call the *local neighborhood*. This set of peers, obtained from the tracker, is unstructured and random, requiring no special join or recovery operations when new peers arrive or existing peers depart. The control traffic required for data exchange is minimal: each peer transmits messages indicating the data blocks they currently possess and messages signaling their interest in the blocks of other peers.

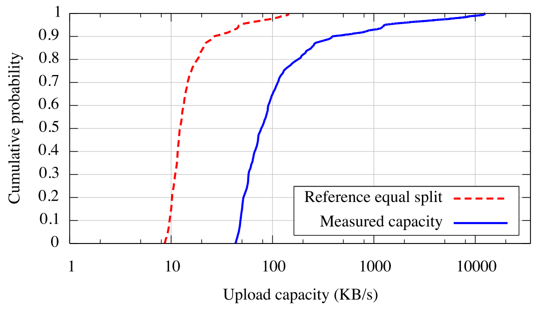


Figure 1: Cumulative distribution of raw bandwidth capacity for BitTorrent peers as well as the "equal split" capacity distribution for active set peers, assuming clients use the reference implementation of BitTorrent.

We refer to the set of peers to which a BitTorrent client is currently sending data as its *active set*. BitTorrent uses a rate-based TFT strategy to determine which peers to include in the active set. Each round, a peer sends data to *unchoked* peers from which it received data most rapidly in the recent past. This strategy is intended to provide positive incentives for contributing to the system and inhibit free-riding. However, clients also send data to a small number of randomly chosen peers who have not "earned" such status. Such peers are said to be *optimistically unchoked*. Optimistic unchokes serve to bootstrap new peers into the TFT game as well as to facilitate discovery of new, potentially better sources of data. Peers that do not send data quickly enough to earn reciprocation are removed from the active set during a TFT round and are said to be *choked*.

Modulo TCP effects and assuming last-hop bottleneck links, each peer provides an equal share of its available upload capacity to peers to which it is actively sending data. We refer to this rate throughout the paper as a peer's *equal split rate*. This rate is determined by the upload capacity of a particular peer and the size of its active set. In the official reference implementation of BitTorrent, active set size is proportional to $\sqrt{\text{upload capacity}}$ (details in Appendix); although in other popular BitTorrent clients, this size is static.

2.2 Measurement

BitTorrent's behavior depends on a large number of parameters: topology, bandwidth, block size, churn, data availability, number of directly connected peers, active TFT transfers, and number of optimistic unchokes. Furthermore, many of these parameters are a matter of pol-

Implementation	Percentage share
Azureus	47%
BitComet	20%
μtorrent	15%
BitLord	6%
Unknown	3%
Reference	2%
Remaining	7%

Table 1: BitTorrent implementation usage as drawn from measurement data.

icy unspecified by the BitTorrent protocol itself. These policies may vary among different client implementations, and defaults may be overridden by explicit user configuration. To gain an understanding of BitTorrent's behavior and the diversity of implementations in the wild, we first conducted a measurement study of live BitTorrent swarms to ascertain client characteristics.

By making use of the opportunistic measurement techniques presented by Madhyastha et al. [14], we gather empirical measurements of BitTorrent swarms and hosts. Our measurement client connected to a large number of swarms and waited for an optimistic unchoke from each unique peer. We then estimated the upload capacity of that client using the multiQ tool [10]. Previous characterizations of end-host capacities of peer-to-peer participants were conducted by Saroiu, et al. [18]. We update these results using more recent capacity estimation tools. We observed 301,595 unique BitTorrent IP addresses over a 48 hour period during April, 2006 from 3,591 distinct ASes across 160 countries. The upload capacity distribution for typical BitTorrent peers is given in Figure 1 along with the distribution of equal split rates that would arise from peers using the reference BitTorrent implementation with no limit on upload rates.

3 Modeling altruism in BitTorrent

In this section, we examine two questions relevant to understanding how incentives impact performance in BitTorrent: how much altruism is present, and what are the sources of altruism? The first question suggests whether or not strategizing is likely to improve performance while the second informs design. Answering these questions for real world swarms is complicated by the diversity of implementations and a myriad of configuration parameters. Here, we take a restricted view and develop a model of altruism arising from our observed capacity distribution and the default parameter settings of the reference implementation of BitTorrent.

We make several assumptions to simplify our analysis and provide a conservative bound on altruism. Because our assumptions are not realistic for all swarms, our modeling results are not intended to be predictive. Rather, our results simply suggest potential sources of altruism and the reasons they emerge in BitTorrent swarms today. We exploit these sources of altruism in the design of our real world strategic client, discussed in Section 4.

- *Representative distribution*: The CDF shown in Figure 1 is for the bandwidth capacity of observed IP addresses over many swarms. The distribution of a typical swarm may not be identical. For instance, high capacity peers tend to finish more quickly than low capacity peers, but they may also join more swarms simultaneously. If they join only a single swarm and leave shortly after completion, the relative proportion of low capacity peers would increase over the lifetime of a swarm.
- *Uniform sizing*: Peers, other than the modified client, use the active set sizing recommended by the reference BitTorrent implementation. In practice, other BitTorrent implementations are more popular (see Table 1) and have different active set sizes. As we will show, aggressive active set sizes tend to decrease altruism, and the reference implementation uses the most aggressive strategy among the popular implementations we inspected. As a result, our model provides a conservative estimate of altruism.
- *No steady state*: Active sets are comprised of peers with random draws from the overall upload capacity distribution. If churn is low, over time TFT may match peers with similar equal split rates, biasing active set draws. We argue in the next section that BitTorrent is slow to reach steady-state, particularly for high capacity peers.
- *High block availability*: Swarm performance is limited by upload capacity, i.e., peers will always be able to find interesting data to download. We find that although the reference BitTorrent implementation is designed to ensure high availability of interesting blocks, in practice, static active set sizing in some clients may degrade block availability for high capacity peers.

These assumptions allow us to model altruism in BitTorrent in terms of the upload capacity distribution only. The model is built on expressions for the probability of TFT reciprocation, expected download rate, and expected upload rate. In this section, we focus on the main insights provided by our model. The precise expressions are listed in detail in the Appendix.

3.1 Tit-for-tat matching time

Since our subsequent modeling results assume that swarms do not reach steady state, we first examine the convergence properties of the TFT strategy used to match peers of similar capacity. By default, the reference BitTorrent client optimistically unchokes two peers every 30 seconds in an attempt to explore the local neighborhood for better reciprocation pairings. Since all peers are

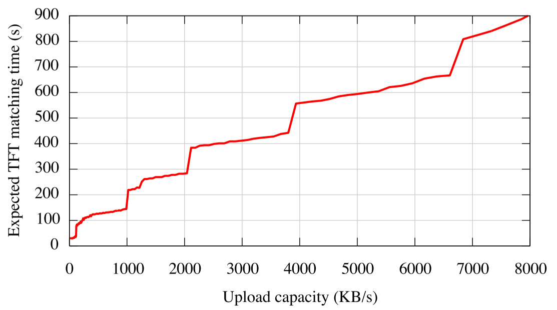


Figure 2: Assuming a peer set of infinite size, the expected time required for a new peer to discover enough peers of equal or greater equal split capacity to fill its active set.

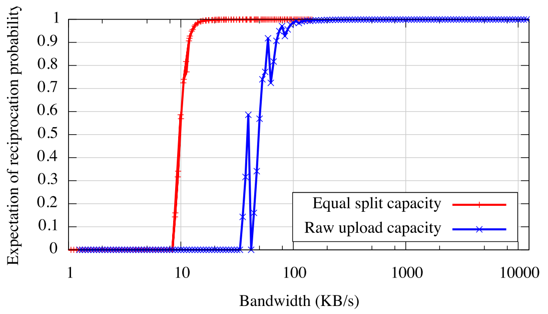


Figure 3: Reciprocation probability for a peer as a function of raw upload capacity as well as reference BitTorrent equal split bandwidth. Reciprocation probability is not strictly increasing in raw rate due to the sawtooth increase in active set size (see Table 2 in Appendix).

performing this exploration concurrently, every 30 seconds a peer can expect to explore two candidate peers and be explored by two candidate peers. Since we know the equal split capacity distribution, we can express the probability of finding a peer with equal or greater equal split capacity—in a given number of 30 second rounds. Taking the expectation and multiplying it by the size of the active set gives an estimate of how long a new peer will have to wait before filling its active set with such peers.

Figure 2 shows this expected time for our observed bandwidth distribution. These results suggest that TFT as implemented does not quickly find good matches for high capacity peers, even in the absence of churn. For example, a peer with 6,400 KB/s upload capacity would transfer more than 4 GB of data before reaching steady state. In practice, convergence time is likely to be even longer. We consider a peer as being "content" with a matching once its equal split is matched or exceeded by a peer. However, one of the two peers in any matching that is not exact will be searching for alternates and switching when they are discovered, causing the other to renew its search. The long convergence time suggests a potential source of altruism: high capacity clients are forced to peer with those of low capacity while searching for better peers via optimistic unchokes.

3.2 Probability of reciprocation

A node Q sends data only to those peers in its active transfer set, reevaluated every 10 seconds. If a peer P sends data to Q at a rate fast enough to merit inclusion in Q's active transfer set, P will receive data during the next TFT round, and we say Q reciprocates with P.

Reciprocation from Q to P is determined by two factors: the rate at which P sends data to Q and the rates at which *other* peers send data to Q. If all other peers in Q's current active set send at rates greater than P, Q will not reciprocate with P.

Figure 3 gives the probability of reciprocation in terms of both raw upload capacity and, more significantly, the equal split rate. The sharp jump in reciprocation probability suggests a potential source of altruism in BitTorrent: equal split bandwidth allocation among peers in the active set. Beyond a certain equal split rate ($\sim$14 KB/s in Figure 3), reciprocation is essentially assured, suggesting that further contribution may be altruistic.

3.3 Expected download rate

Each TFT round, a peer P receives data from both TFT reciprocation and optimistic unchokes. Reciprocation is possible only from those peers in P's active set and depends on P's upload rate, while optimistic unchokes may be received from any peer in P's local neighborhood, regardless of upload rate. In the reference BitTorrent client, the number of optimistic unchoke slots defaults to 2 and is rotated randomly. As each peer unchokes two peers per round, the expected number of optimistic unchokes P will receive is also two for a fixed local neighborhood size.

Figure 4 gives the expected download throughput for peers as a function of upload rate for our observed bandwidth distribution. The sub-linear growth suggests significant unfairness in BitTorrent, particularly for high capacity peers. This unfairness improves performance for the majority of low capacity peers, suggesting that high capacity peers may be able to better allocate their upload capacity to improve their own performance.

3.4 Expected upload rate

Having considered download performance, we turn next to upload contribution. Two factors can control the upload rate of a peer: data availability and capacity limit. When a peer is constrained by data availability, it does not have enough data of interest to its local neighborhood to saturate its capacity. In this case, the peer's upload capacity is wasted and utilization suffers. Because of the dependence of upload utilization on data availability, it is crucial that a client downloads new data at a rate fast

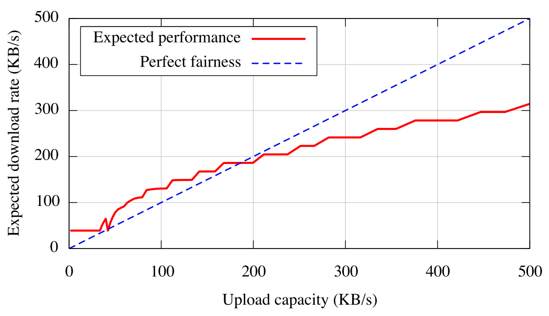


Figure 4: Expectation of download performance as a function of upload capacity. Although this represents a small portion of the spectrum of observed bandwidth capacities, ∼80% of samples are of capacity ≤ 200 KB/s.

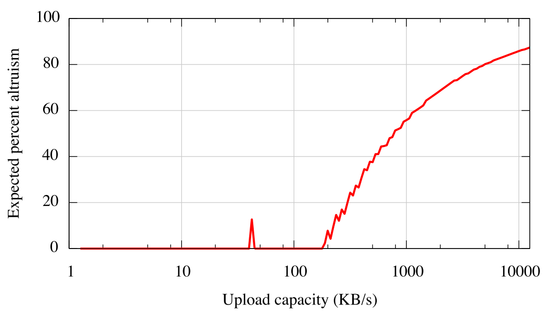


Figure 5: Expected percentage of upload capacity which is altruistic as defined by Equation 5 as a function of rate. The sawtooth increase is due to the sawtooth growth of active set sizing and equal split rates arising from integer rounding (see Table 2).

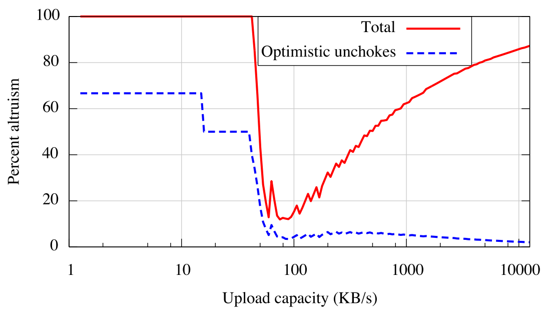


Figure 6: Expected percentage of upload capacity which is altruistic when defined as upload capacity not resulting in direct reciprocation.

enough, so that the client can redistribute the downloaded data and saturate its upload capacity. We have found that indeed this is the case in the reference BitTorrent client because of the square root growth rate of its active set size.

In practice, most popular clients do not follow this dynamic strategy and instead make active set size a configurable, but static, parameter. For instance, the most popular BitTorrent client in our traces, Azureus, suggests a default active set size of four—appropriate for many cable and DSL hosts, but far lower than is required for high capacity peers. We explore the impact of active set sizing further in Section 4.1.

3.5 Modeling altruism

Given upload and download throughput, we have all the tools required to compute altruism. We consider two definitions of altruism intended to reflect two perspectives on what constitutes strategic behavior. We first consider altruism to be simply the difference between expected upload rate and download rate. Figure 5 shows altruism as a percentage of upload capacity under this definition and reflects the asymmetry of upload contribution and download rate discussed in Section 3.3. The second definition is *any* upload contribution that can be withdrawn without loss in download performance. This is shown in Figure 6.

In contrast to the original definition, Figure 6 suggests that *all* peers make altruistic contributions that could be eliminated. Sufficiently low bandwidth peers almost never earn reciprocation, while high capacity peers send much faster than the minimal rate required for reciprocation. Both of these effects can be exploited. Note that low bandwidth peers, despite not being reciprocated, still receive data in aggregate faster than they send data. This is because they receive indiscriminate optimistic unchokes from other users in spite of their low upload capacity.

3.6 Validation

Our modeling results suggest that at least part of the altruism in BitTorrent arises from the sub-linear growth of download throughput as a function of upload rate. We validate this key result using our measurement data. Each time a BitTorrent client receives a complete data block from another peer, it broadcasts a ‘have’ message indicating that it can redistribute that block to other peers. By averaging the rate of have messages over the duration our measurement client observes a peer, we can infer the peer’s download rate. Figure 7 shows this inferred download rate as a function of equal split rate, i.e., the throughput seen by the measurement client when optimistically unchoked. This data is drawn from our measurements and includes 63,482 peers.

These results indicate an even higher level of altruism than that predicted by our model (Figure 4). Note that equal split rate, the parameter of Figure 7, is a conservative lower bound on total upload capacity, shown in Figure 4, since each client sends data to many peers simultaneously. For instance, peers contributing ∼250 KB/s to our measurement client had an observed download rate of 150 KB/s. Our model suggests that such contribution, even when split among multiple peers, should induce a

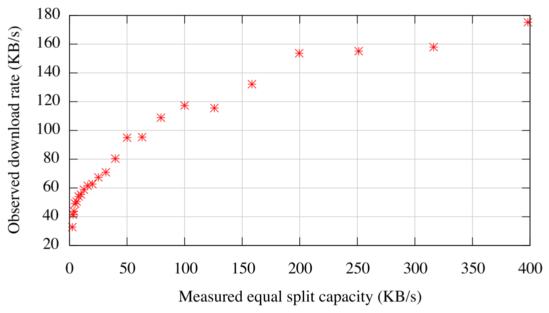


Figure 7: Measured validation of sub-linear growth in download throughput as a function of rate. Each point represents an average taken over all peers with measured equal split capacity in the intervals between points.

download rate of more than 200 KB/s. We believe this underestimate is due to more conservative active set sizes in practice than those assumed in our model.

4 Building *BitTyrant*: A strategic client

The modeling results of Section 3 suggest that altruism in BitTorrent serves as a kind of progressive tax. As contribution increases, performance improves, but not in direct proportion. In this section, we describe the design and implementation of *BitTyrant*, a client optimized for strategic users. We chose to base *BitTyrant* on the Azureus client in an attempt to foster adoption, as Azureus is the most popular client in our traces.

If performance for low capacity peers is disproportionately high, a strategic user can simply exploit this unfairness by masquerading as many low capacity clients to improve performance [4]. Also, by flooding the local neighborhood of high capacity peers, low capacity peers can inflate their chances of TFT reciprocation by dominating the active transfer set of a high capacity peer. In practice, these attacks are mitigated by a common client option to refuse multiple connections from a single IP address. Resourceful peers might be able to coordinate multiple IP addresses, but such an attack is beyond the capabilities of most users. We focus instead on practical strategies that can be employed by typical users.

The unfairness of BitTorrent has been noted in previous studies [2, 5, 7], many of which include protocol redesigns intended to promote fairness. However, a clean-slate redesign of the BitTorrent protocol ignores a different but important incentives question: how to get users to adopt it? As shown in Section 3, the majority of BitTorrent users benefit from its unfairness today. Designs intended to promote fairness globally at the expense of the majority of users seem unlikely to be adopted. Rather than focus on a redesign at the protocol level, we focus on BitTorrent's robustness to strategic behavior and find that strategizing can improve performance in isolation while promoting fairness at scale.

4.1 Maximizing reciprocation

The modeling results of Section 3 and the operational behavior of BitTorrent clients suggest the following three strategies to improve performance.

- *Maximize reciprocation bandwidth per connection*: All things being equal, a node can improve its performance by finding peers that reciprocate with high bandwidth for a low offered rate, dependent only on the other peers of the high capacity node. The reciprocation bandwidth of a peer is dependent on its upload capacity and its active set size. By discovering which peers have large reciprocation bandwidth, a client can optimize for a higher reciprocation bandwidth per connection.
- *Maximize number of reciprocating peers*: A client can expand its active set to maximize the number of peers that reciprocate until the marginal benefit of an additional peer is outweighed by the cost of reduced reciprocation probability from other peers.
- *Deviate from equal split*: On a per-connection basis, a client can lower its upload contribution to a particular peer as long as that peer continues to reciprocate. The bandwidth savings could then be reallocated to new connections, resulting in an increase in the overall reciprocation throughput.

The modeling results indicate that these strategies are likely to be effective. The largest source of altruism in our model is unnecessary contribution to peers in a node's active set. The reciprocation probability shown in Figure 3 indicates that strategically choosing equal split bandwidth can reduce contribution significantly for high capacity peers with only a marginal reduction in reciprocation probability. A peer with equal split capacity of 100 KB/s, for instance, could reduce its rate to 15 KB/s with a reduction in expected probability of reciprocation of only 1%. However, reducing from 15 KB/s to 10 KB/s would result in a decrease of roughly 40%.

The reciprocation behavior points to a performance trade-off. If the active set size is large, equal split capacity is reduced, reducing reciprocation probability. However, an additional active set connection is an additional opportunity for reciprocation. To maximize performance, a peer should increase its active set size until an additional connection would cause a reduction in reciprocation across all connections sufficient to reduce overall download performance.

If the equal split capacity distribution of the swarm is known, we can derive the active set size that maximizes the expected download rate. For our observed bandwidth distribution, Figure 8 shows the download rate as a function of the active set size for a peer with 300 KB/s upload capacity as well as the active set size that maximizes it. The graph also implicitly reflects the sensitivity of recip-

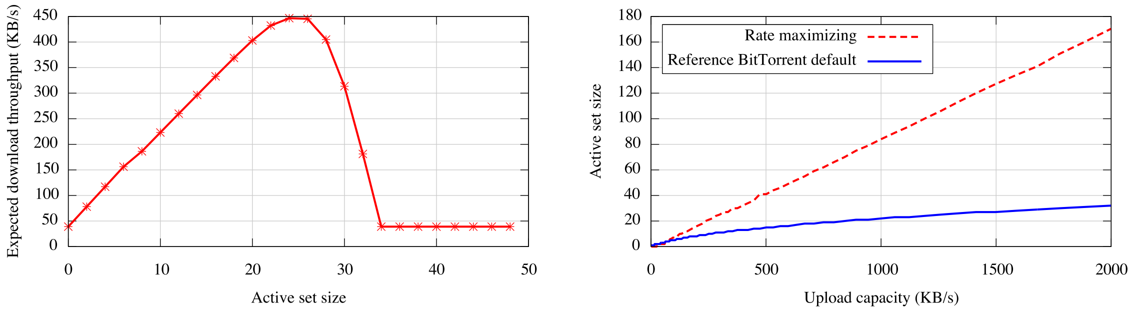


Figure 8: *Left:* The expected download performance of a client with 300 KB/s upload capacity for increasing active set size. *Right:* The performance-maximizing active set size for peers of varying rate. The strategic maximum is linear in upload capacity, while the reference implementation of BitTorrent suggests active size $\sim \sqrt{\text{rate}}$. Although several hundred peers may be required to maximize throughput, most trackers return fewer than 100 peers per request.

rocation probability to equal split rate.

Figure 8 is for a single strategic peer and suggests that strategic high capacity peers can benefit much more by manipulating their active set size. Our example peer with upload capacity 300 KB/s realizes a maximum download throughput of roughly 450 KB/s. However, increasing reciprocation probability via active set sizing is extremely sensitive—throughput falls off quickly after the maximum is reached. Further, it is unclear if active set sizing alone would be sufficient to maximize reciprocation in an environment with several strategic clients.

These challenges suggest that *any* a priori active set sizing function may not suffice to maximize download rate for strategic clients. Instead, they motivate the dynamic algorithm used in *BitTyrant* that adaptively modifies the size and membership of the active set and the upload bandwidth allocated to each peer (see Figure 9).

In both BitTorrent and *BitTyrant*, the set of peers that will receive data during the next TFT round is decided by the unchoke algorithm once every 10 seconds. *BitTyrant* differs from BitTorrent as it dynamically sizes its active set and varies the sending rate per connection. For each peer p, *BitTyrant* maintains estimates of the upload rate required for reciprocation, u_p, as well as the download throughput, d_p, received when p reciprocates. Peers are ranked by the ratio d_p/u_p and unchoked in order until the sum of u_p terms for unchoked peers exceeds the upload capacity of the *BitTyrant* peer.

The rationale underlying this unchoke algorithm is that the best peers are those that reciprocate most for the least number of bytes contributed to them, given accurate information regarding u_p and d_p. Implicit in the strategy are the following assumptions and characteristics:

- The strategy attempts to maximize the download rate for a given upload budget. The ranking strategy corresponds to the value-density heuristic for the knapsack problem. In practice, the download benefit (d_p) and upload cost (u_p) are not known a priori. The update operation dynamically estimates these rates and, in conjunction with the ranking strategy, optimizes download rate over time.
- *BitTyrant* is designed to tap into the latent altruism in most swarms by unchoking the most altruistic peers. However, it will continue to unchoke peers until it exhausts its upload capacity even if the marginal utility is sub-linear. This potentially opens *BitTyrant* itself to being cheated, a topic we return to later.
- The strategy can be easily generalized to handle concurrent downloads from multiple swarms. A client can optimize the aggregate download rate by ordering the d_p/u_p ratios of all connections across swarms, thereby

For each peer p, maintain estimates of expected download performance d_p and upload required for reciprocation u_p.

> Initialize u_p and d_p assuming the bandwidth distribution in Figure 2.
>
> d_p is initially the expected equal split capacity of p.
>
> u_p is initially the rate just above the step in the reciprocation probability.

Each round, rank order peers by the ratio d_p/u_p and unchoke those of top rank until the upload capacity is reached.

$$\underbrace{\frac{d_0}{u_0}, \frac{d_1}{u_1}, \frac{d_2}{u_2}, \frac{d_3}{u_3}, \frac{d_4}{u_4}}_{\text{choose } k \,\mid\, \sum_{i=0}^{k} u_i \,\leq\, cap}, \ldots$$

At the end of each round for each unchoked peer:

> If peer p does not unchoke us: $u_p \leftarrow (1+\delta)u_p$
>
> If peer p unchokes us: $d_p \leftarrow$ observed rate.
>
> If peer p has unchoked us for the last r rounds: $u_p \leftarrow (1-\gamma)u_p$

Figure 9: *BitTyrant* unchoke algorithm

dynamically allocating upload capacity to all peers. User-defined priorities can be implemented by using scaling weights for the d_p/u_p ratios.

The algorithm is based on the ideal assumption that peer capacities and reciprocation requirements are known. We discuss how to predict them next.

Determining upload contributions: The *BitTyrant* unchoke algorithm must estimate u_p, the upload contribution to p that induces reciprocation. We initialize u_p based on the distribution of equal split capacities seen in our measurements, and then periodically update it depending on whether p reciprocates for an offered rate. In our implementation, u_p is decreased by $\gamma = 10\%$ if the peer reciprocates for $r = 3$ rounds, and increased by $\delta = 20\%$ if the peer fails to reciprocate after being unchoked during the previous round. We use small multiplicative factors since the spread of equal split capacities is typically small in current swarms. Although a natural first choice, we do not use a binary search algorithm, which maintains upper and lower bounds for upload contributions that induce reciprocation, because peer reciprocation changes rapidly under churn and bounds on reciprocation-inducing uploads would eventually be violated.

Estimating reciprocation bandwidths: For peers that unchoke the *BitTyrant* client, d_p is simply the rate at which data was obtained from p. Note that we do not use a packet-pair based bandwidth estimation technique as suggested by Bharambe [2], but rather consider the average download rate over a TFT round. Based on our measurements, not presented here due to space limitations, we find that packet-pair based bandwidth estimates do not accurately predict peers' equal split capacities due to variability in active set sizes and end-host traffic shaping. The observed rate over a longer period is the only accurate estimate, a sentiment shared by Cohen [3].

Of course, this estimate is not available for peers that have not uploaded any data to the *BitTyrant* client. In such cases, *BitTyrant* approximates d_p for a given peer p by measuring the frequency of block announcements from p. The rate at which new blocks arrive at p provides an estimate of p's download rate, which we use as an estimate of p's total upload capacity. We then divide the estimated capacity by the Azureus recommended active set size for that rate to estimate p's equal split rate. This strategy is likely to overestimate the upload capacities of unobserved peers, serving to encourage their selection from the ranking of d_p/u_p ratios. At present, this preference for exploration may be advantageous due to the high end skew in altruism. Discovering high end peers is rewarding: between the 95th and 98th percentiles, reciprocation throughput doubles. Of course, this strategy may open *BitTyrant* itself to exploitation, e.g., if a peer rapidly announces false blocks. We discuss how to make *BitTyrant* robust in Sections 4.3 and 5.

4.2 Sizing the local neighborhood

Existing BitTorrent clients maintain a pool of typically 50–100 directly connected peers. The set is sized to be large enough to provide a diverse set of data so peers can exchange blocks without data availability constraints. However, the modeling results of Section 4.1 suggest that these typical local neighborhood sizes will not be large enough to maximize performance for high capacity peers, which may need an active set size of several hundred peers to maximize download throughput. Maintaining a larger local neighborhood also increases the number of optimistic unchokes received.

To increase the local neighborhood size in *BitTyrant*, we rely on existing BitTorrent protocol mechanisms and third party extensions implemented by Azureus. We request as many peers as possible from the centralized tracker at the maximum allowed frequency. Recently, the BitTorrent protocol has incorporated a DHT-based distributed tracker that provides peer information and is indexed by a hash of the torrent. We have increased the query rate of this as well. Finally, the Azureus implementation includes a BitTorrent protocol extension for gossip among peers. Unfortunately, the protocol extension is push-based; it allows for a client to gossip to its peers the identity of its other peers but cannot induce those peers to gossip in return. As a result, we cannot exploit the gossip mechanism to extract extra peers.

A concern when increasing the size of the local neighborhood is the corresponding increase in protocol overhead. Peers need to exchange block availability information, messages indicating interest in blocks, and peer lists. Fortunately, the overhead imposed by maintaining additional connections is modest. In comparisons of *BitTyrant* and the existing Azureus client described in Section 5, we find that average protocol overhead as a percentage of total file data received increases from 0.9% to 1.9%. This suggests that scaling the local neighborhood size does not impose a significant overhead on *BitTyrant*.

4.3 Additional cheating strategies

We now discuss more strategies to improve download performance. We do not implement these in *BitTyrant* as they can be thwarted by simple fixes to clients. We mention them here for completeness.

Exploiting optimistic unchokes: The reference BitTorrent client optimistically unchokes peers randomly. Azureus, on the other hand, makes a weighted random choice that takes into account the number of bytes exchanged with a peer. If a peer has built up a deficit in the number of traded bytes, it is less likely to be picked for optimistic unchokes. In BitTorrent today, we observe that high capacity peers are likely to have trading deficits with most peers. A cheating client can exploit this by disconnecting and reconnecting with a different client identifier, thereby wiping out the past history and increasing its chances of receiving optimistic unchokes, particularly from high capacity peers. This exploit becomes ineffective if clients maintain the IP addresses for all peers encountered during the download and keep peer statistics across disconnections.

Downloading from seeds: Early versions of BitTorrent clients used a seeding algorithm wherein seeds upload to peers that are the fastest downloaders, an algorithm that is prone to exploitation by fast peers or clients that falsify download rate by emitting 'have' messages. More recent versions use a seeding algorithm that performs unchokes randomly, spreading data in a uniform manner that is more robust to manipulation.

Falsifying block availability: A client would prefer to unchoke those peers that have blocks that it needs. Thus, peers can appear to be more attractive by falsifying block announcements to increase the chances of being unchoked. In practice, this exploit is not very effective. First, a client is likely to consider most of its peers interesting given the large number of blocks in a typical swarm. Second, false announcements could lead to only short-term benefit as a client is unlikely to continue transferring once the cheating peer does not satisfy issued block requests.

5 Evaluation

To evaluate *BitTyrant*, we explore the performance improvement possible for a single strategic peer in synthetic and current real world swarms as well as the behavior of *BitTyrant* when used by all participants in synthetic swarms.

Evaluating altruism in BitTorrent experimentally and at scale is challenging. Traditional wide-area testbeds such as PlanetLab do not exhibit the highly skewed bandwidth distribution we observe in our measurements, a crucial factor in determining the amount of altruism. Alternatively, fully configurable local network testbeds such as Emulab are limited in scale and do not incorporate the myriad of performance events typical of operation in the wide-area. Further, BitTorrent implementations are diverse, as shown in Table 1.

To address these issues, we perform two separate evaluations. First, we evaluate *BitTyrant* on real swarms drawn from popular aggregation sites to measure real world performance for a single strategic client. This provides a concrete measure of the performance gains a user can achieve today. To provide more insight into how *BitTyrant* functions, we then revisit these results on PlanetLab where we evaluate sensitivity to various upload rates

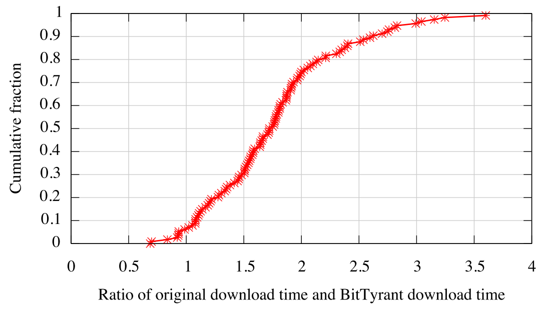


Figure 10: CDF of download performance for 114 real world swarms. Shown is the ratio between download times for an existing Azureus client and *BitTyrant*. Both clients were started simultaneously on machines at UW and were capped at 128 KB/s upload capacity.

and evaluate what would happen if *BitTyrant* is universally deployed.

5.1 Single strategic peer

To evaluate performance under the full diversity of realistic conditions, we crawled popular BitTorrent aggregation websites to find candidate swarms. We ranked these by popularity in terms of number of active participants, ignoring swarms distributing files larger than 1 GB. The resulting swarms are typically for recently released files and have sizes ranging from 300–800 peers, with some swarms having as many as 2,000 peers.

We then simultaneously joined each swarm with a *BitTyrant* client and an unmodified Azureus client with recommended default settings. We imposed a 128 KB/s upload capacity limit on each client and compared completion times. This represents a relatively well provisioned peer for which Azureus has a recommended active set size. A CDF of the ratio of original client completion time to *BitTyrant* completion time is given in Figure 10. These results demonstrate the significant, real world performance boost that users can realize by behaving strategically. The median performance gain for *BitTyrant* is a factor of 1.72 with 25% of downloads finishing at least twice as fast with *BitTyrant*. We expect relative performance gains to be even greater for clients with greater upload capacity.

These results provide insight into the performance properties of real BitTorrent swarms, some of which limit *BitTyrant*'s effectiveness. Because of the random set of peers that BitTorrent trackers return and the high skew of real world equal split capacities, *BitTyrant* cannot always improve performance. For instance, in *BitTyrant*'s worst-performing swarm, only three peers had average equal split capacities greater than 10 KB/s. In contrast, the unmodified client received eight such peers. Total download time was roughly 15 minutes, the typical minimum request interval for peers from the tracker. As a re-

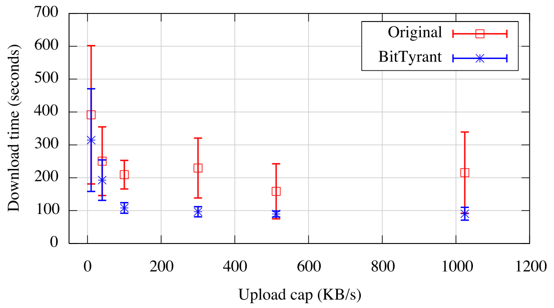


Figure 11: Download times and sample standard deviation comparing performance of a single *BitTyrant* client and an unmodified Azureus client on a synthetic PlanetLab swarm.

sult, *BitTyrant* did not recover from its initial set of comparatively poor peers. To some extent, performance can be based on luck with respect to the set of initial peers returned. More often than not, *BitTyrant* benefits from this, as it always requests a comparatively large set of peers from the tracker.

Another circumstance for which *BitTyrant* cannot significantly improve performance is a swarm whose aggregate performance is controlled by data availability rather than the upload capacity distribution. In the wild, swarms are often hamstrung by the number of peers seeding the file—i.e., those with a complete copy. If the capacity of these peers is low or if the torrent was only recently made available, there may simply not be enough available data for peers to saturate their upload capacities. In other words, if a seed with 128 KB/s capacity is providing data to a swarm of newly joined users, those peers will be able to download at a rate of at most 128 KB/s regardless of their capacity. Because many of the swarms we joined were recent, this effect may account for the 12 swarms for which download performance differed by less than 10%.

These scenarios can hinder the performance of *BitTyrant*, but they account for a small percentage of our observed swarms overall. For most real swarms today, users can realize significant performance benefits from the strategic behavior of *BitTyrant*.

Although the performance improvements gained from using *BitTyrant* in the real world are encouraging, they provide little insight into the operation of the system at scale. We next evaluate *BitTyrant* in synthetic scenarios on PlanetLab to shed light on the interplay between swarm properties, strategic behavior, and performance. Because PlanetLab does not exhibit the highly skewed bandwidth distribution observed in our traces, we rely on application level bandwidth caps to artificially constrain the bandwidth capacity of PlanetLab nodes in accordance with our observed distribution. However, because PlanetLab is often oversubscribed and shares bandwidth equally among competing experiments, not all nodes are capable of matching the highest values from the observed distribution. To cope with this, we scaled by 1/10th both the upload capacity draws from the distribution as well as relevant experimental parameters such as file size, initial unchoke bandwidth, and block size. This was sufficient to provide overall fidelity to our intended distribution.

Figure 11 shows the download performance for a single *BitTyrant* client as a function of rate averaged over six trials with sample standard deviation. This experiment was hosted on 350 PlanetLab nodes with bandwidth capacities drawn from our scaled distribution. Three seeds with combined capacity of 128 KB/s were located at UW serving a 5 MB file. We did not change the default seeding behavior, and varying the combined seed capacity had little impact on overall swarm performance after exceeding the average upload capacity limit. To provide synthetic churn with constant capacity, each node's *BitTyrant* client disconnected immediately upon completion and reconnected immediately.

The results of Figure 11 provide several insights into the operation of *BitTyrant*.

- *BitTyrant* does not simply improve performance, it also provides more consistent performance across multiple trials. By dynamically sizing the active set and preferentially selecting peers to optimistically unchoke, *BitTyrant* avoids the randomization present in existing TFT implementations, which causes slow convergence for high capacity peers (Section 3.1).
- There is a point of diminishing returns for high capacity peers, and *BitTyrant* can discover it. For clients with high capacity, the number of peers and their available bandwidth distribution are significant factors in determining performance. Our modeling results from Section 4.1 suggest that the highest capacity peers may require several hundred available peers to fully maximize throughput due to reciprocation. Real world swarms are rarely this large. In these circumstances, *BitTyrant* performance is consistent, allowing peers to detect and reallocate excess capacity for other uses.
- Low capacity peers can benefit from *BitTyrant*. Although the most significant performance benefit comes from intelligently sizing the active set for high capacity peers (see Figure 8), low capacity peers can still improve performance with strategic peer selection, providing them with an incentive to adopt *BitTyrant*.
- Fidelity to our specified capacity distribution is consistent across multiple trials. Comparability of experiments is often a concern on PlanetLab, but our results suggest a minimum download time determined by the capacity distribution that is consistent across trials spanning several hours. Further, the consistent performance of *BitTyrant* in comparison to unmodi-

fied Azureus suggests that the variability observed is due to policy and strategy differences and not PlanetLab variability.

5.2 Many *BitTyrant* peers

Given that all users have an individual incentive to be strategic in current swarms, we next examine the performance of *BitTyrant* when used by all peers in a swarm. We consider two types of *BitTyrant* peers: strategic and selfish. Any peer that uses the *BitTyrant* unchoking algorithm (Figure 9) is strategic. If such a peer also withholds contributing excess capacity that does not improve performance, we say it is both strategic and selfish. *BitTyrant* can operate in either mode. Selfish behavior may arise when users participate in multiple swarms, as discussed below, or simply when users want to use their upload capacity for services other than BitTorrent.

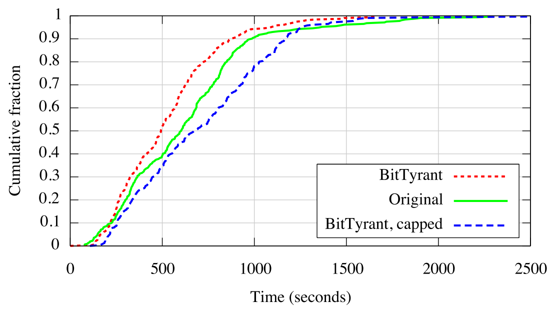
We first examine performance when all peers are strategic, i.e., use *BitTyrant* while still contributing excess capacity. Our experimental setup included 350 PlanetLab nodes with upload capacities drawn from our scaled distribution simultaneously joining a swarm distributing a 5 MB file with combined seed capacity of 128 KB/s. All peers departed immediately upon download completion. Initially, we expected overall performance to degrade since high capacity peers would finish quickly and leave, reducing capacity in the system. Surprisingly, performance improved and altruism increased. These results are summarized by the CDFs of completion times comparing *BitTyrant* and the unmodified Azureus client in Figure 12. These results are consistent with our model. In a swarm where the upload capacity distribution has significant skew, high capacity peers require many connections to maximize reciprocation. *BitTyrant* reduces bootstrapping time and results in high capacity peers having higher utilization earlier, increasing swarm capacity.

Although *BitTyrant* can improve performance, such improvement is due only to more effective use of altruistic contribution. Because *BitTyrant* can detect the point of diminishing returns for performance, these contributions can be withheld or reallocated by selfish clients. Users may choose to reallocate capacity to services other than BitTorrent or to other swarms, as most peers participate in several swarms simultaneously [7]. While all popular BitTorrent implementations support downloading from multiple swarms simultaneously, few make any attempt to intelligently allocate bandwidth among them. Those that do so typically allocate some amount of a global upload capacity to each swarm individually, which is then split equally among peers in statically sized active sets. Existing implementations cannot accurately detect when bandwidth allocated to a given swarm should be reallocated to another to improve performance. In contrast, *BitTyrant*'s unchoking algorithm transitions naturally from single to multiple swarms. Rather than allocate bandwidth among *swarms*, as existing clients do, *BitTyrant* allocates bandwidth among *connections*, optimizing aggregate download throughput over all connections for all swarms. This allows high capacity *BitTyrant* clients to effectively participate in more swarms simultaneously, lowering per-swarm performance for low capacity peers that cannot.

To model the effect of selfish *BitTyrant* users, we repeated our PlanetLab experiment with the upload capacity of all high capacity peers capped at 100 KB/s, the point of diminishing returns observed in Figure 11. A CDF of performance under the capped distribution is shown in Figure 12. As expected, aggregate performance decreases. More interesting is the stable rate of diminishing returns *BitTyrant* identifies. As a result of the skewed bandwidth distribution, beyond a certain point peers that contribute significantly more data do not see significantly faster download rates. If peers reallocate this altruistic contribution, aggregate capacity and average performance are reduced, particularly for low capacity peers. This is reflected in comparing the performance of single clients under the scaled distribution (Figure 11) and single client performance under the scaled distribution when constrained (Figure 12). The average completion time for a low capacity peer moves from 314 to 733 seconds. Average completion time for a peer with 100 KB/s of upload capacity increases from 108 seconds to 190.

While *BitTyrant* can improve performance for a single swarm, there are several circumstances for which its use causes performance to degrade.

- If high capacity peers participate in many swarms or otherwise limit altruism, total capacity per swarm decreases. This reduction in capacity lengthens download times for all users of a single swarm regardless of contribution. Although high capacity peers will see an increase in *aggregate* download rate across many swarms, low capacity peers that cannot successfully compete in multiple swarms simultaneously will see a large reduction in download rates. Still, each individual peer has an incentive to be strategic as their performance improves relative to that of standard clients, even when everyone is strategic or selfish.
- New users experience a lengthy bootstrapping period. To maximize throughput, *BitTyrant* unchokes peers that send fast. New users without data are bootstrapped by the excess capacity of the system only. Bootstrapping time may be reduced by reintroducing optimistic unchokes, but it is not clear that selfish peers have any incentive to do so.
- Peering relationships are not stable. *BitTyrant* was designed to exploit the significant altruism that exists in



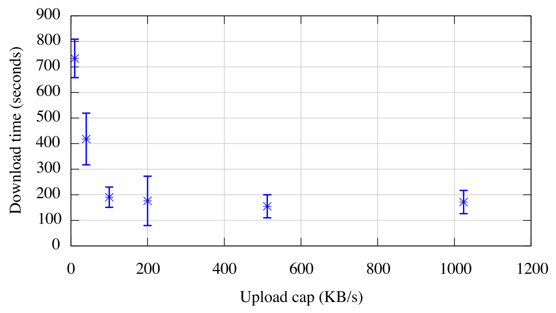


Figure 12: *Left:* CDFs of completion times for a 350 node PlanetLab experiment. *BitTyrant* and the original, unmodified client assume all users contribute all of their capacity. Capped *BitTyrant* shows performance when high capacity, selfish peers limit their contribution to the point of diminishing returns for performance. *Right:* The impact of selfish *BitTyrant* caps on performance. Download times at all bandwidth levels increase (cf. Figure 11) and high capacity peers gain little from increased contribution. Error bars give sample standard deviation over six trials.

BitTorrent swarms today. As such, it continually reduces send rates for peers that reciprocate, attempting to find the minimum rate required. Rather than attempting to ramp up send rates between high capacity peers, *BitTyrant* tends to spread available capacity among many low capacity peers, potentially causing inefficiency due to TCP effects [16].

To work around this last effect, *BitTyrant* advertises itself at connection time using the Peer ID hash. Without protocol modification, *BitTyrant* peers recognize one another and switch to a block-based TFT strategy that ramps up send rates until capacity is reached. *BitTyrant* clients choke other *BitTyrant* peers whose block request rates exceeds their send rates. By gradually increasing send and request rates to other *BitTyrant* clients, fairness is preserved while maximizing reciprocation rate with fewer connections. In this way, *BitTyrant* provides a deployment path leading to the conceptually simple strategy of block-based TFT by providing a short-term incentive for adoption by all users—even those that stand to lose from a shift to block-based reciprocation.

We do not claim that *BitTyrant* is strategyproof, even when extended with block-based TFT, and leave open for future work the question of whether further strategizing can be effective. However, a switch to block-based TFT among mutually agreeing peers would place a hard limit on altruism and limit the range of possible strategies.

6 Related work

Modeling and analysis of BitTorrent's current incentive mechanism and its effect on performance has seen a large body of work since Cohen's [3] seminal paper. Our effort differs from existing work in two fundamental ways. First is the *conclusion*: we refute popular wisdom that BitTorrent's incentive mechanism makes it robust to strategic peer behavior. Second is the *methodology*: most existing studies consider small or simulated settings that poorly capture the diversity of deployed BitTorrent clients, strategic peer behavior, peer capacities, and network conditions. In contrast, we explore BitTorrent's strategy space with our implementation of a strategic client and evaluate it using analytical modeling, experiments under realistic network conditions, and testing in the wild.

The canonical TFT strategy was first evaluated by Axelrod [1], who showed using a competition that the strategy performs better than other submissions when there are many repeated games, persistent identities, and no collusion. Qiu and Srikant [17] specifically study BitTorrent's rate-based TFT strategy. They show that if peers strategically limit their upload bandwidth (but split it equally) while trying to maximize download, then, under some bandwidth distributions, the system converges to a Nash equilibrium where all peers upload at their capacity. These results might lead one to believe that BitTorrent's incentive mechanism is robust as it incentivizes users to contribute their entire upload capacities. Unfortunately, our work shows that BitTorrent fails to attain such an equilibrium for typical file sizes in swarms with realistic bandwidth distributions and churn, which *BitTyrant* exploits through strategic peer and rate selection.

Bharambe et al. [2] simulate BitTorrent using a synthetically generated distribution of peer upload capacities. They show the presence of significant altruism in BitTorrent and propose two alternate peer selection algorithms based on (i) matching peers with similar bandwidth, and (ii) enforcing TFT at the block level, a strategy also proposed by [9]. Fan et al. propose strategies for assigning rates to connections [5], which when adopted by all members of a swarm would lead to fairness and minimal altruism. The robustness of these mechanisms to strategic peer behavior is unclear. More importantly, these proposals appear to lack a convincing evolution path—a peer adopting these strategies today would severely hurt its download throughput as the majority of deployed conformant clients will find such a peer unattractive. In contrast, we demonstrate that *BitTyrant* can drastically reduce altruism while improving performance for a single strategic client today, incenting its adoption.

Shneidman et al. [19] identify two forms of strategic manipulation based on Sybil attacks [4] and a third based on uploading garbage data. Liogkas et al. [12] propose downloading only from seeds and also identify an exploit based on uploading garbage data. Locher et al. investigate similar techniques, i.e., ignoring rate limits of tracker requests to increase the number of available peers and connecting to as many peers as possible [13]. However, there exist straightforward fixes to minimize the impact of such "byzantine" behavior. A third exploit by Liogkas et al. involves downloading only from the fastest peers, but the strategy does not take into account the upload contribution required to induce reciprocation. In contrast, *BitTyrant* maximizes download per unit of upload bandwidth and can drastically reduce its upload contribution by varying the active set size and not sharing its upload bandwidth uniformly with active peers.

Hales and Patarin [8] argue that BitTorrent's robustness is not so much due to its TFT mechanism, but more due to human or sociological factors that cause swarms with a high concentration of altruistic peers to be preserved over selfish ones. They further claim that releasing selfish clients into the wild may therefore not degrade performance due to the underlying natural selection. Validating this hypothesis requires building and releasing a strategic and selfish client—one of our contributions.

Massoulie and Vojnovic [15] model BitTorrent as a "coupon replication" system with a particular focus on efficiently locating the last few coupons. One of their conclusions is that altruism is not necessary for BitTorrent to be efficient. However, their study does not account for strategic behavior on the part of peers.

Other studies [2, 7, 11] have pointed out the presence of significant altruism in BitTorrent or suggest preserving it [11]. In contrast, we show that the altruism is not a consequence of BitTorrent's incentive mechanism and can in fact be easily circumvented by a strategic client.

7 Conclusion

We have revisited the issue of incentive compatibility in BitTorrent and arrived at a surprising conclusion: although TFT discourages free riding, the bulk of BitTorrent's performance has little to do with TFT. The dominant performance effect in practice is altruistic contribution on the part of a small minority of high capacity peers. More importantly, this altruism is not a consequence of TFT; selfish peers—even those with modest resources—can significantly reduce their contribution and yet improve their download performance. BitTorrent works well today simply because most people use client software as-is without trying to cheat the system.

Although we have shown that selfishness can hurt swarm performance, whether or not it will do so in practice remains unclear. The public release of *BitTyrant* provides a test. Perhaps users will continue to donate their excess bandwidth, even after ensuring the maximum yield for that bandwidth. Perhaps users will behave selfishly, causing a shift to a completely different design with centrally enforced incentives. Perhaps strategic behavior will induce low bandwidth users to invest in higher bandwidth connections to compensate for their worse performance, yielding better overall swarm performance in the long run. Time will tell. These uncertainties leave us with the still open question: do incentives build robustness in BitTorrent?

The *BitTyrant* source code and distribution are publicly available at:

`http://BitTyrant.cs.washington.edu/`

Acknowledgments

We thank our shepherd, Jinyang Li, and the anonymous reviewers for their comments. This work was supported by NSF CNS-0519696 and the ARCS Foundation.

References

[1] R. Axelrod. *The Evolution of Cooperation*. Basic Books, 1985.

[2] A. Bharambe, C. Herley, and V. Padmanabhan. Analyzing and Improving a BitTorrent Network's Performance Mechanisms. In *Proc. of INFOCOM*, 2006.

[3] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. of IPTPS*, 2003.

[4] J. R. Douceur. The Sybil attack. In *Proc. of IPTPS*, 2002.

[5] B. Fan, D.-M. Chiu, and J. Liu. The Delicate Tradeoffs in BitTorrent-like File Sharing Protocol Design. In *Proc. of ICNP*, 2006.

[6] GNU Scientific Library. `http://www.gnu.org/software/gsl/`.

[7] L. Guo, S. Chen, Z. Xiao, E. Tan, X. Ding, and X. Zhang. Measurements, analysis, and modeling of BitTorrent-like systems. In *Proc. of IMC*, 2005.

[8] D. Hales and S. Patarin. How to Cheat BitTorrent and Why Nobody Does. Technical Report UBLCS 2005-12, Computer Science, University of Bologna, 2005.

[9] S. Jun and M. Ahamad. Incentives in BitTorrent induce free riding. In *Proc. of P2PECON*, 2005.

[10] S. Katti, D. Katabi, C. Blake, E. Kohler, and J. Strauss. MultiQ: Automated detection of multiple bottleneck capacities along a path. In *Proc. of IMC*, 2004.

[11] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest First and Choke Algorithms are Enough. In *Proc. of IMC*, 2006.

[12] N. Liogkas, R. Nelson, E. Kohler, and L. Zhang. Exploiting BitTorrent for fun (but not profit). In *Proc. of IPTPS*, 2006.

Label	Definition	Meaning
ω	2	Number of simultaneous optimistic unchokes per peer
λ	80	Local neighborhood size (directly connected peers)
$b(r)$	Figure 1	Probability of upload capacity rate r
$B(r)$	$\int_0^r b(r)dr$	Cumulative probability of a upload capacity rate r
$\text{active}(r)$	$\lfloor\sqrt{0.6r}\rfloor - \omega$	Size (in peers) of the active transfer set for upload capacity rate r
$\text{split}(r)$	$\frac{r}{\text{active}(r)+\omega}$	Per-connection upload capacity for upload capacity rate r
$s(r)$	Figure 1	Probability of an equal split rate r using mainline $\text{active}(r)$ sizing
$S(r)$	$\int_0^r s(r)dr$	Cumulative probability of an equal-split rate r

Table 2: Functions used in our model and their default settings in the official BitTorrent client.

[13] T. Locher, P. Moor, S. Schmid, and R. Wattenhofer. Free Riding in BitTorrent is Cheap. In *Proc. of HotNets*, 2006.

[14] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani. iPlane: An information plane for distributed services. In *Proc. of OSDI*, 2006.

[15] L. Massoulié; and M. Vojnović. Coupon replication systems. *SIGMETRICS Perform. Eval. Rev.*, 33(1):2–13, 2005.

[16] R. Morris. TCP behavior with many flows. In *Proc. of ICNP*, 1997.

[17] D. Qiu and R. Srikant. Modeling and performance analysis of BitTorrent-like peer-to-peer networks. In *Proc. of SIGCOMM*, 2004.

[18] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of Multimedia Computing and Networking*, 2002.

[19] J. Shneidman, D. Parkes, and L. Massoulié. Faithfulness in internet algorithms. In *Proc. of PINS*, 2004.

A Modeling notes

All numerical evaluation was performed with the GSL numerics package [6]. Refer to Section 3 for assumptions and Table 2 for definitions.

Upload / download: Probability of reciprocation for a peer P with upload capacity r_P from Q with r_Q:

$$\text{p\_recip}(r_P, r_Q) = 1 - (1 - S(r_P))^{\text{active}(r_Q)} \tag{1}$$

Expected reciprocation probability for capacity r:

$$\text{recip}(r) = \int b(x)\text{p\_recip}(r, x)dx \tag{2}$$

Expected download and upload rate for capacity r:

$$\text{D}(r) = \text{active}(r)\left[\int b(x)\text{p\_recip}(r,x)\text{split}(x)dx\right] + \omega\left[\int b(x)\text{split}(x)dx\right] \tag{3}$$

$$\text{U}(r) = \min\big(r, (\text{active}(r) + \omega)\ \text{D}(r)\big) \tag{4}$$

Altruism: Altruism when defined as the difference between upload contribution and download reward

$$\text{altruism\_gap}(r) = \max\big(0,\ \text{U}(r) - \text{D}(r)\big) \tag{5}$$

Altruism per connection when defined as upload contribution not resulting in direct reciprocation.

$$\text{altruism\_conn}(r) = \int \Big(b(x)\big((1 - \text{p\_recip}(r,x))\text{split}(r) + \text{p\_recip}(r,x)\max(0, \text{split}(r) - \text{split}(x))\big)\Big)dx \tag{6}$$

Total altruism not resulting in direct reciprocation.

$$\text{altruism}(r) = (\text{active}(r) + \omega)\text{altruism\_conn}(r) \tag{7}$$

Convergence: Probability of a peer with rate r discovering matched TFT peer in n iterations:

$$c(r, n) = 1 - S(r)^{n\ 2\omega} \tag{8}$$

Time to populate active set with matched peers given upload capacity r. Note, $s = \text{split}(r)$, and T = 30s is the period after which optimistic unchokes are switched.

$$\text{convergence\_time}(r) = T \cdot \text{active}(r)\Big(c(s,1) + \sum_{n=2}^{\infty} n\ c(s,n) \prod_{i=1}^{n-1}(1 - c(s,i))\Big) \tag{9}$$

Unchoke probability: The distribution of number of optimistic unchokes is binomial with success probability $\frac{\omega}{\lambda}$. Because overhead is low, $\lambda \gg \text{active}(r)$ in *BitTyrant*, we approximate $\lambda - \text{active}(r)$ by λ. The expected number of optimistic unchokes per round is ω.

$$Pr[\text{unchokes} = x] = \binom{\lambda}{x}\left(\frac{\omega}{\lambda}\right)^x\left(1 - \frac{\omega}{\lambda}\right)^{(\lambda - x)} \tag{10}$$

$$\therefore E[\text{unchokes}] = \lambda\frac{\omega}{\lambda} = \omega$$

Exploiting Similarity for Multi-Source Downloads Using File Handprints

Himabindu Pucha, David G. Andersen, Michael Kaminsky
Purdue University, Carnegie Mellon University, Intel Research Pittsburgh

Abstract

Many contemporary approaches for speeding up large file transfers attempt to download chunks of a data object from multiple sources. Systems such as BitTorrent quickly locate sources that have an exact copy of the desired object, but they are unable to use sources that serve similar but non-identical objects. Other systems automatically exploit cross-file similarity by identifying sources for each chunk of the object. These systems, however, require a number of lookups proportional to the number of chunks in the object and a mapping for each unique chunk in every identical and similar object to its corresponding sources. Thus, the lookups and mappings in such a system can be quite large, limiting its scalability.

This paper presents a hybrid system that provides the best of both approaches, locating identical *and* similar sources for data objects using a constant number of lookups and inserting a constant number of mappings per object. We first demonstrate through extensive data analysis that similarity does exist among objects of popular file types, and that making use of it can sometimes substantially improve download times. Next, we describe *handprinting*, a technique that allows clients to locate similar sources using a constant number of lookups and mappings. Finally, we describe the design, implementation and evaluation of Similarity-Enhanced Transfer (SET), a system that uses this technique to download objects. Our experimental evaluation shows that by using sources of similar objects, SET is able to significantly out-perform an equivalently configured BitTorrent.

1 Introduction

Extensive research over the past several decades has explored many techniques to improve data transfer speed and efficiency. Despite this effort, bulk data transfers often remain slow for a variety of reasons. First, of course, receivers may be bandwidth-limited. Second, the source or sources may be unable to saturate the receiver's bandwidth. Third, congestion or failures in the "middle" of the network may slow the transfer.

Downloading data from multiple sources simultaneously is a popular technique to speed transfers when the receiver is not the bottleneck. Many peer-to-peer content distribution systems use this idea, for example. These systems employ two basic strategies to locate the sources of data: per-file and per-chunk.

In a per-file system, receivers locate other sources of the *exact* file they are downloading in $O(1)$ lookups. These systems, exemplified by BitTorrent [7], Gnutella [1] and ChunkCast [6], typically use a global location service. Unfortunately, as we explore further in Section 2, the performance of file transfers using these systems is often unacceptably slow, with users requiring hours or even days to download content.

In a per-chunk system, receivers locate sources for individual pieces, or chunks, of the desired file. Since any given chunk in a file might appear in several other files, finding sources for each chunk can yield a much larger number of *similar* sources. The cost, however, is performing $O(N)$ lookups, one for each of the N chunks in the file the receiver is trying to download. Moreover, such systems also require a mapping between every unique chunk in the identical and similar files and their corresponding sources, i.e., $O(N)$ mappings per object. Examples of per-chunk systems are CFS [9] and Shark [3].

In this paper, we present Similarity-Enhanced Transfer (SET)—a hybrid system that provides the best of both approaches, locating both identical *and* similar sources for data chunks using $O(1)$ lookups and by inserting $O(1)$ mappings per file. We justify this approach by demonstrating that (a) cross-file similarity exists in real Internet workloads (i.e., files that people are actually downloading on today's file-sharing networks); (b) we can find similar sources in $O(1)$ lookups; (c) the extra overhead of locating these sources does not out-weigh the benefit of using them to help saturate the recipient's available bandwidth. Indeed, exploiting similar sources can significantly improve download time.

The three contributions of this paper are centered around these points. First, we present a detailed similarity analysis of 1.7 TB of data fetched from several active file-sharing networks. These data represent a variety of file types, with an emphasis on multimedia files often ignored by previous similarity studies. Our results show that significant cross-file similarity exists in the files that are frequently transferred on these networks. By exploiting this similarity, receivers can locate several times the number of potential sources (Section 3).

The second contribution of this paper is a technique to locate similar sources to the file being downloaded using only $O(1)$ lookups. This technique, which we term *handprinting*, is a novel use of deterministic sampling. Sources insert a fixed number of hashes into a global database; receivers look up their own set of hashes in this database to find sources of similar files. System-wide parameters determine the amount of similarity receivers can detect (e.g., all files with *x*% similarity to the target file) and with what probability they can detect it (Section 4).

Third, to demonstrate the benefit of this approach to multi-source downloads, we built a prototype system that uses handprinting to locate sources of similar files (Section 5). Our results show that the overhead of our approach is acceptable (roughly 0.5% per similar file). Without using similar sources, the prototype meets or exceeds BitTorrent's performance. When we enable downloads from similar sources, the system finds and uses these sources to greatly improve transfer speeds (Section 6).

2 Motivation: Slow Downloads

With fast, asymmetric broadband connections, receivers are frequently unable to saturate their available bandwidth during downloads, even when using existing multi-source peer-to-peer download programs. For example, in 2003, Gummadi et al. found that 66.2% of downloads failed, and that the median transfer time in Kazaa for files over 100 MB was *over one day* [13]. In 2002, 75% of Gnutella peers had under 1 megabit of upstream capacity [28].

Unfortunately, our experience suggests that this situation has not improved in the intervening years. Figure 1 shows the CDF of throughput achieved while downloading 6208 large files from popular file-sharing networks (the details of our measurement study appear in Section 3.1). The median transfer achieved under 10 Kbit/s of average throughput, and the 90th percentile only managed 50 Kbit/s, despite running these experiments from an extremely well-connected academic network.

While they are a step in the right direction, today's file-sharing applications clearly have room for improvement. We hypothesize that this improvement can come from giving these applications additional sources from which to download—specifically, by allowing them to download from sources with files similar to their target file.

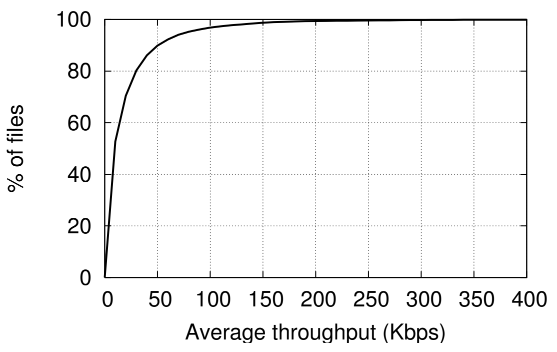


Figure 1: Throughput observed in 6208 file downloads from file-sharing networks.

3 Similarity

We care about *exploitable* similarity that can be used to speed transfers. In most multi-source systems, including the one we propose here, the smallest unit of data transfer is a chunk. Chunks can have a static or variable size. To be able to detect similarity between mis-aligned objects, we define the chunk boundaries using Rabin fingerprinting.[1] Exploitable similarity means that another file shares entire chunks with the target file that the receiver is trying to download. We therefore define similarity as the fraction of chunks that are shared between two files.[2]

Prior studies have found exploitable similarity in filesystems (particularly source code and compiler output) [19], email attachments [31, 10], HTML documents and web objects [18, 11], and operating system and application software [8]. The amount of similarity ranged from a few percent for compressed files in filesystems, to 15% for Web objects, to 30% for email.

We were concerned, however, that such similarity would not arise in heavily-compressed multimedia files, which comprise the bulk of traffic on file-sharing networks. To our surprise, we found considerable similarity in these files. This similarity arose for many reasons; examples include:

- MP3 music files with identical sound content but different header bytes (artist and title metadata or headers from encoding programs) were 99% similar.
- Movies and trailers in different languages were often 15% or more similar.

[1]Rabin fingerprinting, as used in systems such as LBFS [19] determines chunk boundaries by examining the hash of a sliding window over the file data, and declaring a chunk boundary when the *N* lowest bits of the hash are equal to some fixed value (e.g., zero). Using this mechanism, small changes to the content *within* a chunk, including adding or removing bytes, will generally not change the boundaries themselves.

[2]For reasons we discuss further in Section 4, if the files are of different size, we define similarity as the smaller of the pairwise similarities.

- Media files with apparent transmission or storage errors differed in a single byte or small string of bytes in the middle of the file.
- Identical content packaged for download in different ways (e.g., a torrent with and without a README file) were almost identical.

Note that in all of these cases, the modified files (and their hashes) differ from the originals; traditional, per-file systems that look only for identical files cannot find and use the modified versions as additional sources for the original (desired) object. Furthermore, note that our similarity study measures similarity among files exactly as they are offered online for download (e.g., we do not uncompress, recode, etc. media files).

In the rest of this section, we explore the feasibility of using the exploitable similarity among files to improve the performance of multi-source transfers. We propose two metrics to capture the benefit of using similar files. We apply these metrics to a large volume of data downloaded from file-sharing networks to show the effective increase in the number of sources that a client of one of these networks might experience.

3.1 Data Collection

We examined several large software collections as well as 1.7 TB of files downloaded from popular file-sharing networks. The results from analyzing the software archives were similar to those in previous studies. Due to space constraints, we chose not to include them here, but to instead focus on the more novel study of multimedia files.

File sources. We collected files from the eDonkey and Overnet networks using the MLDonkey client [2] for three months between November 2006 and February 2007. We modified the client to record the IP addresses of the sources for files and to not upload blocks.[3]

Selecting files to download. We could only download a fraction of the files available on these file-sharing networks, so the choice of what to download was important. We intentionally sampled a subset of files that were likely to be similar to each other and that we expected people to be interested in downloading.

To do so, we selected groups of unique files returned as search results for a given query. Our hypothesis is that files in these groups are more likely to be similar than would two files chosen completely at random. This approach necessarily provides only a lower bound on similarity that really exists in the file-sharing network (sampling more files would reveal more similarity). To find files that people were interested in downloading, we categorize our search terms as "popular" or "unpopular" (tail). We chose popular terms from recent Billboard top ten audio tracks and top ten box office results. We chose unpopular audio search terms from top song collections between 1990 and 2000, and video terms from one of the authors' personal Netflix history list. We selected terms that resulted in fewer than 50 (but at least one) search result. Our final search terms consisted of 15 popular and 11 unpopular song title/artist pairs plus 14 popular and 12 unpopular movie titles. We ordered the search results by decreasing number of sources and downloaded as many files as possible.

Cleaning the data. 2567 of the files (78% of them audio files) we downloaded contained garbage. A common source of garbage is services such as Overpeer, which disseminate fake files to discourage sharing copyrighted content [16]. To eliminate obviously incorrect results, we filtered downloads by first compressing with rzip. If the "media" files compressed more than 20%, we eliminated them. Second, we passed MP3s through an MP3-to-WAV converter; if the converter exited with an error, we eliminated the file. Spot checks of the eliminated and retained files suggested that this method produced few false positives and negatives. These garbage files are excluded from all results in this paper—we analyzed only the remaining 6208 files.

3.2 Summary of Similarity Study

Table 1 summarizes the data we collected, broken down by the top three file types. MP3 files had a high median similarity: if one MP3 file was similar to another, it was likely to be nearly identical. In contrast, video files generally had lower similarity. One reason we found for such similarity was multiple versions of the same video in different languages. Finally, we note that the median number of similar files is lower for video files due, in part, to the dataset having fewer such files to examine.

3.3 Quantifying the Benefits of Similarity

The basic question we would like to answer is how much does exploiting the observed similarity among files speed up a data transfer. We address this question in two stages. The first step, described in this section, is analytical. We define two *parallelism metrics* and apply them to the data we downloaded. The metrics quantify, on real data, the similarity in the data and the resulting speedup in a transfer from using that similarity. The second step, described in the following sections, is to understand the practicality of using similar sources through the design, implementation, and evaluation of a system that finds and uses such sources for faster multi-source downloads.

[3]BitTorrent's tit-for-tat mechanism meant that we could not effectively download files using it without also being a source for the files we downloaded.

File Type	Count	Median Size (MB)	Median Avg. Throughput (Kbps)	Median Max. Throughput (Kbps)	Median Identical Sources	Median Similarity (%)	Median # Similar Files
MP3	3725	5.3	5.4	31.2	1.1	99.6	25
AVI	1624	699.7	28.8	220.0	6.7	22.9	3
MPEG	533	692.3	22.5	169.6	5.1	99.0	1

Table 1: Similarity in different types of media. Max. throughput refers to the maximum throughput observed during the course of a file download. Median similarity is the median across files with non-zero similarity; the last column shows the median number of files that were similar to each file of this type.

The parallelism metrics attempt to quantify how many different sources a receiver can draw from to download a particular object. One metric assumes limited parallelism ("conservative") and the other assumes perfect parallelism ("optimistic"). This analysis assumes that all sources serve chunks at the same rate and that chunks are the same size.

Both metrics produce similar results when files are either extremely similar or extremely dissimilar. They capture the decreasing benefit of having many sources for only a small subset of the chunks (because the few sources for the remaining chunks will be the bottleneck):

- Two sources that each have almost all of the chunks of the file provide a parallelism approaching 2. Three such sources provide a parallelism of nearly 3, and so on.
- If there are many sources for one chunk in the file, but only one source for all of the remaining chunks, the parallelism is only negligibly over 1.

The two metrics differ in how they treat sources with moderate similarity. Imagine that one source has all of the blocks of the file and another source has half of the blocks. The optimistic metric assumes that a receiver could obtain half the blocks from one source and half from the other, producing a parallelism of two. The conservative metric instead accounts for the loss of flexibility in this scenario compared to two identical sources, and assigns it a value of $1\frac{1}{3}$, for reasons we discuss below.

The optimistic metric assumes that the client can download from all sources in parallel with an optimal choice of which chunk to download from which source. Under this optimal assignment of chunks to sources, each source serves C_s chunks out of the total number of chunks C. The time to download the file, therefore, is determined only by the maximum C_s, the largest number of chunks the receiver must download from any single source. The parallelism, then, is $\frac{C}{max\ C_s}$.

The conservative parallelism metric states that a chunk can be downloaded in time inversely proportional to the number of sources for that chunk.[4] Let:

$$\begin{aligned} C &= \text{The number of chunks in the file} \\ S_i &= \text{The number of sources for chunk } i \\ \frac{1}{S_i} &= \text{The time to download chunk } i \end{aligned}$$

Conservative parallelism is then the ratio of the original download time C (one unit of time per chunk) over the parallel download time:

$$\text{Parallelism} = \frac{C}{\sum_{i=1}^{C} \frac{1}{S_i}}$$

The conservative metric assumes *limited* parallelism. Chunks are downloaded one-at-a-time, but any given chunk is downloaded from all available sources in parallel. In the example above with one complete source and one source with half of the chunks, the chunk download time would be $\frac{C}{\frac{C}{2}\cdot\frac{1}{2}+\frac{C}{2}\cdot\frac{1}{1}} = \frac{4}{3}$. If instead these two sources each had a complete copy, the parallelism would be $C/\left(C\cdot\frac{1}{2}\right) = 2$.

Which metric is correct depends on the capacities of the sources and the network topology. Shared bottlenecks and differing source capacities will make the conservative metric more accurate; independent sources with similar capacities, plus intelligent assignment of chunks to sources, will favor the parallel metric.

The analysis that follows uses both metrics to provide a better picture of the potential benefits. We examine *parallelism gain*, the ratio of parallelism with similar sources to parallelism without similar sources.

3.3.1 Analysis Method

To compute the available parallelism, every 10 minutes during a transfer, we logged both the active IP addresses from which our clients actually downloaded a file as well as the set of addresses returned during a search for the file. We combined the search results by IP address: if a host serves F files that could be useful for a download,

[4] In our system the smallest granularity for download is a chunk, but in aggregate parallelizing within a chunk is equivalent to parallelizing between chunks that are available from the same number of sources.

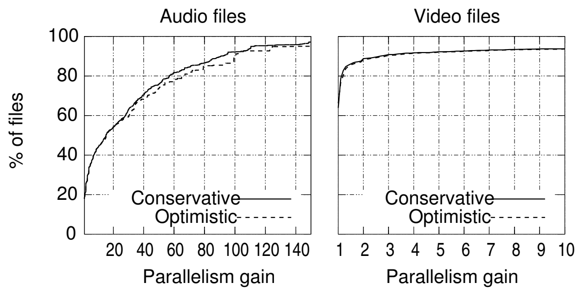


Figure 2: Optimistic and conservative parallelism gain with 16 KB chunk size. Note that the x-axes differ.

the parallelism metrics treat it as single source serving the union of useful chunks across all F files.

We define the set of sources for a file by choosing the active IPs from one random interval during the file transfer. (The results were the same using other choices for the sources: active IPs vs. search IPs, and one random interval vs. the union of all IPs observed during the transfer. The choice did not matter because parallelism gain is *relative*: an increase from 2 identical sources to 4 identical plus similar sources is equivalent to an increase from 20 to 40.)

3.3.2 Optimistic vs. Conservative Parallelism

Figure 2 shows the CDF of parallelism gain for audio and video files split into 16 KB chunks with Rabin fingerprinting. This graph shows three important points. First, 20% of the audio files and 65% of video files were completely dissimilar to the other files we downloaded (they had no exploitable similarity). Second, as expected, the parallelism is higher for the dataset with more samples (audio). With a larger sample size, the study would have discovered a larger number of similar sources for the video files, but (unless our sampling was in some way biased), it would not have discovered a greater degree of similarity among those sources.

3.3.3 Chunk Size and Parallelism

Figure 3 shows the CDF of parallelism gain with chunk sizes that vary from 2 KB to 128 KB. Smaller chunk sizes take advantage of finer-grained similarity between files, but larger chunk sizes impose lower overhead. We set our chunk size to 16 KB, which imposes less than 1% per-chunk overhead in our implementation for identifying and requesting the chunks, but gains most of the parallelism.

3.3.4 Popularity and Parallelism

Figure 4 shows that there is slightly more parallelism gain for popular files than for unpopular files. We attribute this difference primarily to finding a smaller number of "variants" of the less popular content. The parallelism gain for both categories, however, is still significant and can be successfully exploited for objects in both categories.

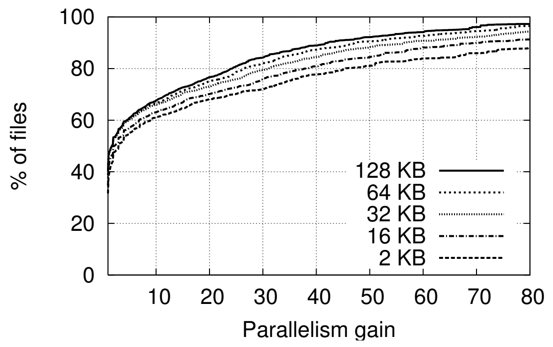


Figure 3: Conservative parallelism gain as chunk size is varied for media files using Rabin fingerprinting.

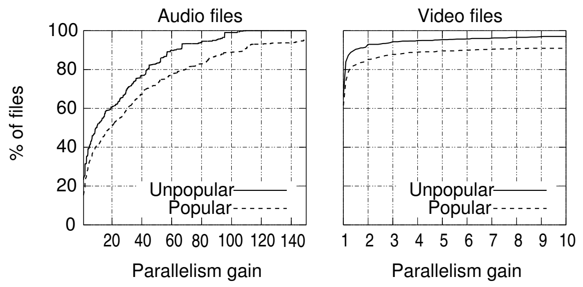


Figure 4: Conservative parallelism gain for files of different popularity.

In summary, many files available on popular file-sharing networks have exploitable similarity that could be used to increase substantially the parallelism available for transfers. The majority of transfers are slow (Figure 1), and these slow transfers have available significant parallelism when using similar files. We therefore believe that allowing clients to use these similar sources could significantly reduce their download time, if only they had a way to do so.

4 Handprinting to Find Similarity

This section presents handprinting, our technique for finding useful sources of shared content. Handprinting is inspired by earlier techniques ("shingling," "fingerprinting," and deterministic sampling) often used to detect similarity between files (e.g., for clustering search results [4] or detecting spam [33]). Our contribution is the adaptation and analysis of these techniques to efficiently locate sources of *exploitable* similarity during file transfer; prior work used them purely as a similarity *metric*.

Handprinting first divides each file into a series of chunks, $C_1 \cdots C_N$, obtained through Rabin fingerprinting.[5] Next, it computes the hash (fingerprint) of each chunk, and takes a deterministic sample of these fingerprints. To sample, handprinting sorts the hashes in lexicographic order, selects the first k hashes (the handprint), and inserts them into a global lookup table. To find similar files, a receiver obtains the chunk hashes for its desired file and searches for entries for any of the first k hashes. The algorithm detects similar files if they collide on any of the k chunks. This process is illustrated in Figure 5, and we analyze the probability of its success below.

4.1 Quantifying Discovery Probability

Consider two files A and B, composed of N_A and N_B chunks, respectively. A and B have m distinct chunks in common (the shaded chunks in Figure 5).

We assume that the hash of a chunk is deterministic (the hash of the same data produces the same value), but that the actual value is effectively random. What is the probability that handprinting detects two shared files with particular values of N_A, N_B, m, k?

Because the chunk hashes are sorted, the handprint will contain shared chunks from the two files *in the same order*. Thus, if the handprints of both files A and B contain at least one shared chunk hash, that chunk hash is guaranteed to be the same chunk in both files A and B. This is unlike randomly selecting k chunk hashes, in which even if the handprints of both files contain a shared chunk, it may not be the same chunk, which thus has a lower probability of intersection. The probability of detection p, then, is simply the probability that the handprint of each file includes one or more shared chunks:

$$\begin{aligned} p &= P(\geq 1 \text{ shared chunk from A}) \cdot \\ &\quad\; P(\geq 1 \text{ shared chunk from B}) \end{aligned}$$

Assuming that the file is large enough to sample with replacement,[6] the probability of choosing at least one shared chunk from a file is:

$$\begin{aligned} P(\text{pick shared}) &= 1 - P(\text{all } k \text{ chunks not shared}) \\ &= 1 - (P(\text{one chunk not shared}))^k \\ &\geq 1 - \left(1 - \frac{m}{N}\right)^k \end{aligned}$$

Thus, the total probability of detecting that the files share chunks is:

$$p \;\geq\; \left(1 - \left(1 - \frac{m}{N_A}\right)^k\right) \cdot \left(1 - \left(1 - \frac{m}{N_B}\right)^k\right)$$

[5]We use Rabin fingerprinting only to determine the chunk boundaries, not to fingerprint the chunks.

[6]A pessimistic assumption; sampling without replacement increases the chances of having at least one collision, but the assumption is reasonable for the large file transfers that handprinting targets.

4.1.1 Defining similarity

If the files are the same size, the probability of detection depends only on k and on the *fraction* of chunks in the files that are shared ($\frac{m}{N}$). This fraction is the *similarity* of the two files, denoted s.

Similarity is more complicated when the two files are of different size. If they share m chunks, is the similarity $\frac{m}{N_A}$, $\frac{m}{N_B}$, or a combination? To simplify analysis, we define the similarity as the minimum of these two similarities. As a result, the probability we established for detecting similarity becomes a slightly more pessimistic lower bound, but the inequality remains correct.

This definition is not merely a simplification, however, since it presents an important limitation of handprinting: Given a target file, it is easier to find shared chunks in files *smaller* than the target than in files larger than the target. With the same number of shared chunks, smaller files will have a larger similarity value than larger files. In practice, however, most of the cases of file similarity that we have observed occur between files of similar size.

4.1.2 How many chunk samples?

The value of k should balance a high chance of detecting similar files with the overhead of performing queries. We can select k to have a certain probability of detection p for files with similarity s:

$$p \;\geq\; \left(1 - (1-s)^k\right)^2$$

solving for k:

$$k \;\geq\; \frac{\log\left(1 - \sqrt{p}\right)}{\log(1-s)}$$

Thus, for example, $k = 28$ provides at least a 90% chance ($p \geq 0.9$) of finding a file that shares 10% or more of its chunks ($s \geq 0.1$) with the target file. Table 2 shows the values of k required to detect 90% of files with a varying similarity threshold. Handprinting with $k = 30$ successfully found over 99% of files in our data set with over 10% similarity. (Recall that for files in our data set that shared chunks, the median similarity was higher than 10%. The probability of finding such files was therefore higher than our 90% target.)

4.1.3 Handprints versus random chunk selection

Handprinting substantially improves the probability of finding similar files versus a naive random chunk selection mechanism. In the limit, consider a scheme that selects one chunk at random from each file. The probability that this scheme will work is the probability of selecting a shared chunk from the first file ($\frac{m}{N}$) and also selecting the

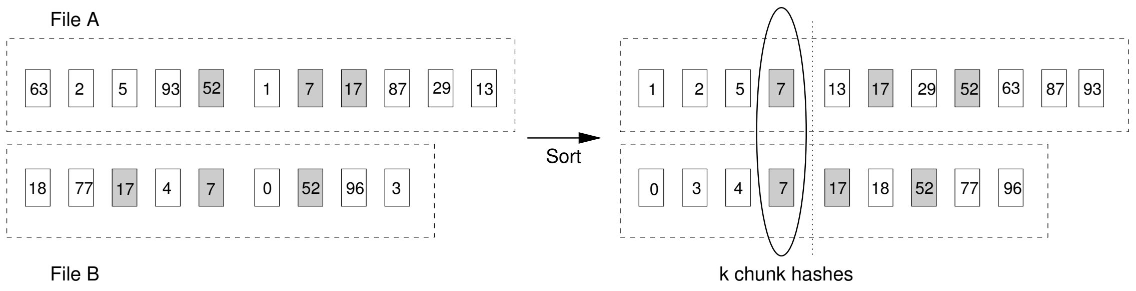


Figure 5: Handprinting. By sorting the chunk hashes before insertion, handprinting imposes a consistent order on hashes so that if a set of one or more shared (shaded) hashes are chosen from each file, the sets will intersect.

Target Similarity (s)	k
90%	1.29
50%	4.29
10%	28.19
5%	57.90
1%	295.55

Table 2: Number of chunk samples for p = 0.9 as similarity is varied.

same chunk from the second file ($\frac{1}{N}$), or $\frac{m}{N^2}$. If the files share 10 out of 100 chunks, the probability of detecting them using the random scheme is 0.001; using handprints, the probability is 0.01, an order of magnitude better.

4.2 Similarity Threshold and Parallelism

Handprinting finds files with a particular threshold similarity. How does this affect the parallelism from the previous section? The answer depends on the distribution of similarity in the files. Figure 6 shows the parallelism with different thresholds, from using all files to using only those that are 90% similar. The left graph shows the curve for audio files, which have a high median similarity. As a result, the available parallelism for these files does not depend greatly on the threshold. The right graph shows the parallelism for video files, which rely more on being able to find less-similar files. We choose a 10% similarity threshold as a generally applicable value that does not impose excess overhead.

5 Design and Implementation

This section describes three aspects of the design of SET. First, we examine how the system determines handprints, how it inserts them into a global lookup table, and how receivers query this table to discover similar sources. Next, we describe the surrounding multi-source framework that we built to download files using handprinting. Finally, we briefly discuss the implementation of SET.

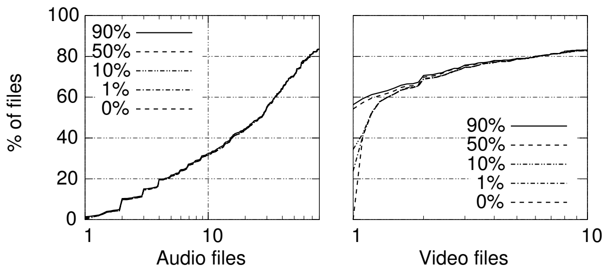


Figure 6: Conservative parallelism as the similarity threshold is varied for smaller audio files and video files. Note that the x-axis of the two graphs differs. The five lines on the left graph completely overlap.

5.1 Locating sources with handprinting

SET allows receivers to download objects from multiple sources, which contain either exact or similar objects, using a constant number of lookups. Every object in SET is identified by its Object ID (OID), a collision-resistant hash of the object. To download a file, the receiver must locate the sources for the actual file it wishes to download (*exact sources*) and for the other files that share chunks with the target file (*similar sources*). We assume the receiver already has a complete list of chunk hashes for the desired object. (For example, such hashes are included as part of the BitTorrent "torrent" file.) The receiver uses these hashes to fetch the individual chunks from the sources it discovers.

For every object that a source makes available, it inserts a set of mappings into a global lookup table. This lookup table could be a Distributed Hash Table (DHT), or a service run by a single organization such as Napster. The lookup table contains two types of mappings: (1) chunk hash → OID, and (2) OID → source.

Mapping 1 associates the hash of a particular chunk in the object with the object identifier for that object. There are exactly *k* mappings that point to each OID, as defined in Section 4. These mappings allow receivers to locate objects that are similar to the one they want to download.

Mapping 2 associates the OID of an object with the

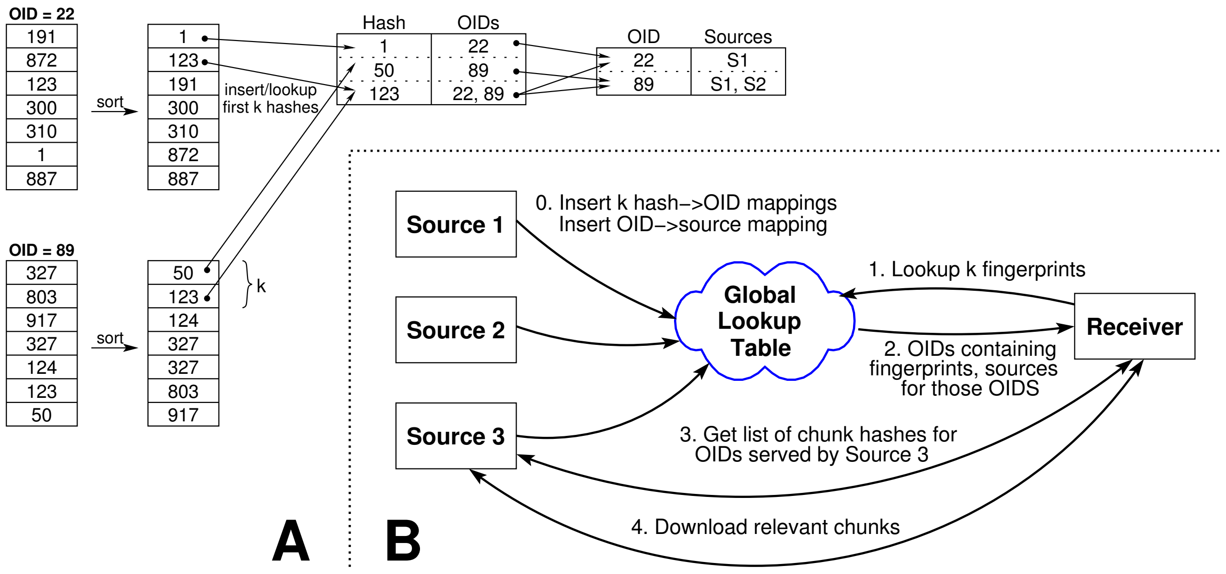


Figure 7: A—Mapping in SET. Mapping 1 links chunk hashes to OIDs; mapping 2 links OIDs to sources of those OIDs. B—SET design overview.

sources for that object. The lookup table contains one such mapping for each source that serves the corresponding object. If three sources serve OID *x*, then three OID-to-source mappings will exist in the lookup table (but still only *k* chunk to OID mappings, since the selected chunks will always be the same for a given OID). The mappings are shown in Figure 7-A.

Given the list of chunk hashes, the receiver computes the object's handprint (the *k* chunk hashes), and looks each hash up in the global lookup table. The result of this query is a list of OIDs that are exploitably similar to the desired file (they share one or more large chunks of data). For each OID, the receiver queries the lookup table to determine the list of potential sources. Finally, the receiver must query one source for each OID to obtain the list of hashes for that OID. At this point, the receiver knows a set of sources that have or are in the process of downloading each of the chunks in the object the receiver wishes to download. This process is illustrated in Figure 7-B.

This process requires a constant number of lookups regardless of the file size: *k* queries to the distributed lookup table to determine the similar files, plus 2× the number of similar OIDs (first to find potential sources for each OID and second to determine the list of hashes for each similar OID). SET limits the number of similar OIDs to a constant (in our implementation, 30), preferring OIDs that matched more chunk hashes if it has an excess number of choices.

Given the list of potential sources for each chunk in the desired file, the receiver can begin downloading the chunks in parallel from the sources.

5.2 Retrieving Chunks

The literature is rich with optimizations for faster ways to download chunks from multiple sources in "swarming" or "mesh" transfers [7, 14, 5, 29, 20]. Our focus in this paper is on evaluating the benefits from handprinting, which we believe apply to many of these systems. We therefore implement a set of capabilities similar to those in BitTorrent for deciding which sources to use for which chunks, though we plan in the future to incorporate more advanced optimizations into SET.

Maintaining an up-to-date list of sources. New sources appear for chunks in two ways:

1. A new network source arrives
2. An already found source obtains a new chunk.

SET periodically queries the global lookup table for new sources for the OIDs it has discovered, and for new OIDs that are similar to its target file. This low-rate periodic refresh is particularly important when downloading large files, which may take sufficient time that many sources may arrive or depart during the download.

On a finer timescale, SET directly queries other clients downloading exact and similar files to retrieve a bitmap of which chunks the clients have. Because they may not have received the entire file, we term these other clients *partial* sources. These partial sources insert mappings into the global table just like complete sources do.[7]

[7]BitTorrent uses asynchronous notifications that a new hash has arrived. While the overhead from querying the entire bitmap is not a bottleneck in our evaluation, we plan to adopt asynchronous notifications to reduce overhead when transferring extremely large files.

Source selection heuristics. SET uses the *rarest-random* strategy [14], selecting chunks uniformly at random from the rarest chunks. It places chunks in a list with other chunks available from the same number of sources, and processes these lists in order. SET also implements an "endgame mode" [15] similar to BitTorrent's. To avoid waiting excessively for slow sources at the end of the transfer, SET requests all outstanding but not yet complete chunks from an additional randomly selected source.

5.3 Implementation

SET is implemented as a set of modules within the Data-Oriented Transfer modular framework [31].

SET implementation. DOT determines an OID as the SHA-1 hash of a file, and a *descriptor list* as the set of chunk hashes for a file. Internally, SET is added to DOT as two new modules. The first is a *storage plugin* that sits between DOT and the disk. When the client receives an object, the SET storage plugin computes the handprint for the object using the descriptor list handed to it from DOT and inserts it into the distributed lookup table. This insertion allows other nodes to locate the data objects cached by a particular node.

The second module is a *transfer plugin*. DOT uses transfer plugins to send and receive chunks across the network. Transfer plugins implement a simple API:

```
get_descriptors(oid, hints)
get_chunk(descriptor, hints)
```

Hints tell the transfer plugin which sources it should attempt to contact to retrieve the descriptor list or the data chunk. The SET transfer plugin sits between DOT and its usual RPC-based transfer plugin. When DOT first requests an object from the SET transfer plugin, the plugin examines the descriptor list returned from the sender, computes the object's handprint, and queries the global lookup table to find similar sources. The plugin then assigns sources to each chunk by setting the chunk's hints according to the source selection heuristics described above. The multi-source transfer plugin then uses these hints to download the actual chunks.

Changes to DOT. SET is one of the first complex transfer plugins developed under DOT, and it exposed two limitations of DOT's original interface. To correct these, we first added to the transfer plugin API a `notify_descriptors(oid, descriptors)` method that lets transfer plugins associate the descriptors they are fetching with the OID they belong to. In the original DOT API, transfer plugins would have only received a chunk request, with no way to find sources for the chunk from similar files.

Next, we added a local and remote `get_bitmap` call to determine which chunks of an object were available at a node. While a DOT plugin could determine this information locally by querying for each descriptor separately, we felt that the efficiency gain justified adding a specialized interface.

For efficiency, we also changed DOT's representation of OIDs and descriptors from 40 byte hex-encoded ASCII strings to 20 byte binary objects. While the original design favored ease of debugging and tracing with a human-readable string, SET retrieves several descriptor lists for each object it downloads, increasing the overhead imposed by the prior ASCII representation.

Distributed Lookup Table. SET inserts and looks up map entries through an RPC put/get interface. The interface supports either OpenDHT [26] (a public, shared distributed hash table that runs on PlanetLab), or our simplified, centralized implementation (*cDHT*). Our evaluation uses cDHT to isolate the results from performance variation due to OpenDHT.

6 Evaluation

Our evaluation of SET examines whether our design and implementation is able to effectively discover and use additional sources of similar objects. The evaluation uses a mix of synthetic data and the data collected for the similarity analysis described in Section 3. We evaluate SET using simplified topologies in Emulab [32] to examine different aspects of its performance in isolation, and in several deployments on PlanetLab [23] ranging from 9–50 nodes. To ensure that the benefits achieved by taking advantage of similar sources are not simply compensating for flaws in our implementation, we compare the performance of SET to that of BitTorrent where appropriate. For the comparison, we disabled BitTorrent's tit-for-tat mechanism to allow it to send at full speed to all clients.

The results of our evaluation are promising: SET matches or exceeds BitTorrent's performance on the same topologies without using similarity. Using even modest amounts of similarity substantially improves SET's performance, handily outperforming the stock BitTorrent client.

Despite being pleased with our system's performance, we note that we do not present this comparison to show that our multi-source downloading is *better* than that of BitTorrent or other protocols. Our contributions are the design and evaluation of efficient mechanisms for discovering similar sources, not any particular optimization in conventional per-file peer-to-peer transfers. Under many circumstances, it is likely that research peer-to-peer systems such as Bullet Prime [14], ChunkCast [6], or CoBlitz [22] could outperform our prototype when SET is not using similarity. Rather, we present the comparison to show that SET's performance is in line with that of other modern download clients and that these systems too could benefit from using handprinting.

6.1 Using similar sources significantly improves transfer speeds

The experiments described in this section are run on Emulab with five receivers attempting to download from one origin source with a variable number of similar sources. Each node is connected to a 100 Mbit/sec "core" via an access link whose size varies by the experiment. To isolate the causes of slowdowns, the access links are configured so that the core is never the bottleneck.

The Emulab experiments compare SET to BitTorrent across four different network configurations. In each configuration, all nodes have the same speed access link. The configurations are:

Slow DSL	384 Kbit/s up, 1500 Kbit/s down, 40 ms RTT
Fast DSL	768 Kbit/s up, 3000 Kbit/s down, 40 ms RTT
T1	1500 Kbit/s symmetric, 40 ms RTT
High BDP	(High bandwidth delay product) 9800 Kbit/s symmetric, 400 ms RTT

For each network configuration, we measure download time for four scenarios:

BT	BitTorrent, no similar sources possible
SET	SET, no similar sources
SET10	SET, low similarity (one 10% similar source)
SET15	SET, modest similarity (three 15% similar sources)
SET90	SET, high similarity (three 90% similar sources)

In the first two scenarios, the five downloaders can download only from each other and from the origin. This provides a baseline comparison of SET and BitTorrent. The next three scenarios add increasing numbers of similar sources to see if SET can use them effectively. We examine sources that are 10%, 15%, and 90% similar. 10% is SET's target minimum similarity. 15% and 90% are the median similarity for video and audio files from our data set, respectively.

Figures 8, 9 and 10 show the average, maximum, and minimum download times for clients for 4 MB, 10 MB and 50 MB files in these scenarios over two runs.[8] SET slightly outperforms BitTorrent without using similar sources. By using similar sources, SET can perform up to

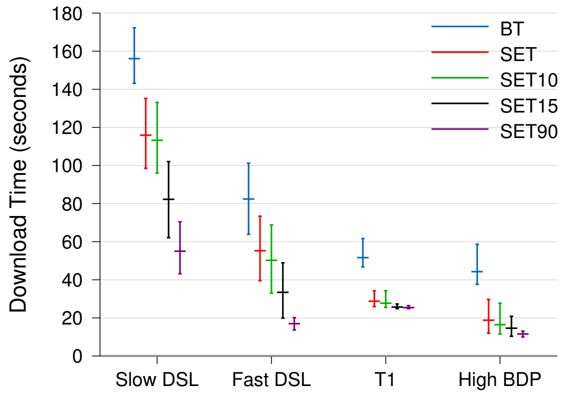


Figure 8: Transfer times for a 4 MB file on Emulab for each transfer method over four network configurations. Within a configuration, the lines appear in the same order as in the legend (BT, SET, SET10, SET15, SET90).

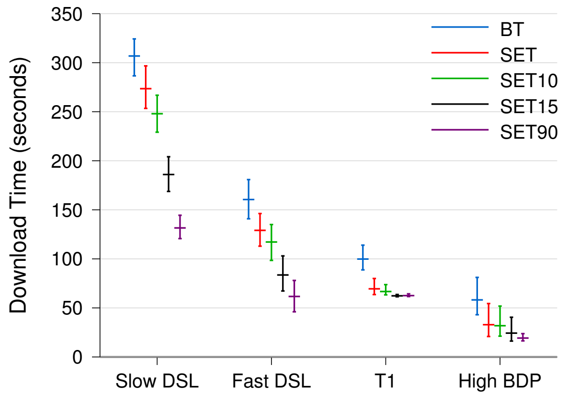


Figure 9: Transfer times for 10 MB file on Emulab

three times faster. Using only a single 10% similar source reduces the average download time by 8%, showing the benefits that arise from exploiting even small amounts of similarity. Similarly, three 15% similar sources reduce the download time by 30% on average. Thus, using larger numbers of similar sources provides increasing benefits. Similar improvements occur across all of the file sizes we examined.

6.2 SET performs well in the real world

We ran the initial experiments (described above) on Emulab in order to isolate the performance effects of varying individual parameters such as file similarity, capacity, and number of sources. In this section, we describe several analogous experiments that we ran on PlanetLab to confirm whether or not SET performs on real networks in the same way it does on an emulated one. These experiments also have five clients downloading from one origin and a variable number of similar sources, except where noted.

[8]We choose 50 MB to facilitate comparability with CoBlitz [20], Shark [3], and Bullet Prime [14].

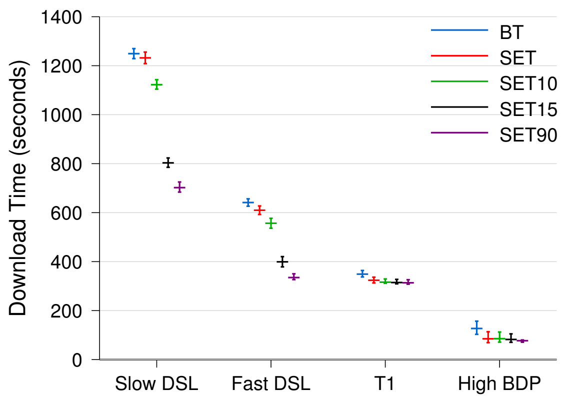


Figure 10: Transfer times for 50 MB file on Emulab

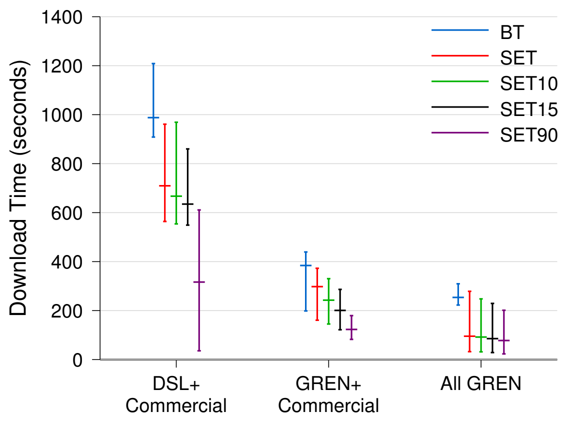


Figure 11: Transfer times for a 50 MB file on PlanetLab for each transfer method over the three configurations.

As in the previous experiments, we find that additional sources of similarity help (and that SET is able to efficiently locate them) to the point that the receiver's available bandwidth becomes the bottleneck. We show this by measuring download time in three scenarios:

- **GREN**: Nine nodes drawn from sites on the Global Research and Education Networks (Internet2, Abilene, etc.). These nodes are extremely well-connected.
- **Mixed**: Nine nodes drawn from commercial and smaller educational networks. This set of nodes is more representative of well-connected commercial sites. There are few enough GREN nodes that most of the transfers occur over commercial Internet paths.
- **DSL+Commercial**: Eight DSL-connected PlanetLab nodes and commercial sites. In this experiment, the similar sources that are added are all moderately well-connected commercial sites, and the clients are mostly DSL connected. This experiment had only four clients because of the limited number of extremely slow clients available.[9]

Figure 11 shows the performance of SET in these PlanetLab scenarios when transferring a 50 MB file. The numbers presented are the average, min, and max of all of the clients across two runs. Several conclusions are apparent from these graphs.

First, when running on the high-bandwidth GREN nodes, the benefits of additional sources are minimal, because senders can already saturate receivers. There is a small reduction in the maximum transfer time from these additional sources, but we suspect this time is no different from that of using multiple TCP connections to a single sender. The larger difference between BT and SET is a combination of 1) SET's ability to immediately begin downloading from the origin before contacting a tracker (or equivalent), and 2) the interpreted BitTorrent client requiring slightly more CPU time under contention on the PlanetLab nodes.

[9]Only three PlanetLab DSL nodes were usable during the times we conducted our experiments.

Second, when running on the more constrained commercial and DSL nodes, using additional sources provides benefit similar to that observed in the Emulab experiments (though with increased variance, as expected).

The huge improvement in transfer times when using three 90% similar sources with the DSL clients (left-hand cluster of lines in Figure 11, rightmost line) arises because one of the similar sources has a faster commercial network connection. When the fast source has only 15% of the blocks, it modestly improves the transfer time, but the overall transfer remains constrained by the slowest 85% of the chunks. However, when the fast source has a 90% similar file, the fast clients can rapidly retrieve most of their chunks from the fast source. While these improvements are in-line with what one would expect, they also show that SET is able to operate effectively when the sources and clients have a diverse range of available bandwidth.

6.3 Scaling with additional similar sources

How well does SET scale as the system has more sources of similar content? In this experiment, the receiver attempts to download a 10 MB file. There is one origin source and a varying number of different 90% similar sources. We examine the same four network configurations used previously (Slow DSL, Fast DSL, T1, and High BDP).

Figure 12 shows two important properties of SET. First, it effectively uses additional, similar sources to speed transfers when the sources are the bottleneck. For the slow

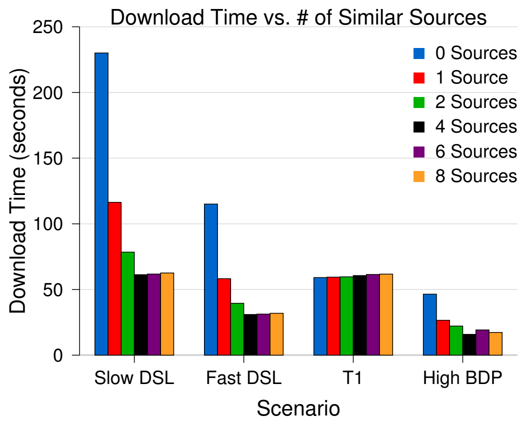


Figure 12: Scaling with a variable number of similar sources and one exact source

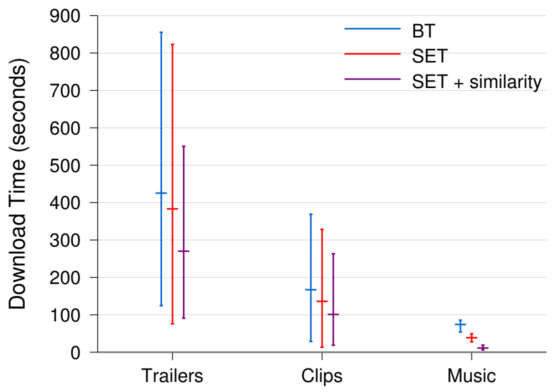


Figure 13: Transfer times for real scenarios on PlanetLab for each transfer method.

DSL scenario, the benefits increase until the receiver's capacity is saturated by $\frac{1500}{384} \approx 4$ sources. The fast DSL behaves similarly, while the symmetric T1 link can be saturated by a single source. As expected, using similar sources also helps when a single TCP connection is window limited and unable to saturate the high bandwidth delay product link.

Second, using similar sources adds minimal overhead in the symmetric case when a single source can saturate the receiver's capacity (the right side of the graph for the T1). The transfer time increases from 59.08 seconds with only the original source to 61.70 seconds with eight (unneeded) similar sources. In general, SET incurs a roughly 0.5% overhead per additional similar file it draws from, because SET must retrieve the full descriptor list for each file, and it must perform periodic bitmap queries from these sources. In practice, we expect the benefits from finding additional sources to outweigh the small overhead required to take advantage of them.

6.4 Scenarios from File-Sharing Networks

We conclude by examining three scenarios drawn from our measurement study of file-sharing networks. For each of these scenarios, we map the sources and clients randomly to a pool of 50 nodes containing all available DSL and commercial nodes on PlanetLab along with randomly chosen GREN nodes.

Different-length movie trailers: Ten receivers attempt to download a 55 MB file from one origin source. They are helped by 21 other sources of a 33 MB file that shares 47% of the target file's blocks.

Minimally-similar movie clips: 14 receivers download a 17 MB file from one origin. They are helped by 32 sources of a 19 MB file that is 15% similar to the target.

Unpopular music: Four receivers download an 11 MB MP3 file from one origin. They are helped by four other clients that have an MP3 file that is 99% similar to the target file.

SET effectively reduces the receivers' download times in these scenarios (Figure 13). Using a 47% similar file reduces the movie trailer download time by 36% and 30% compared to BitTorrent and SET without similarity, respectively. In the second scenario, the 15% similar file reduced the movie clip transfer times by 36% and 26%. Unsurprisingly, the common case of a 99% similar music file was greatly improved using similarity, reducing its transfer time by 71% over SET without similarity.

7 Applicability of SET

From our evaluation, we know that SET works well when there are adequate amounts of exploitable similarity and the original senders cannot saturate the receivers' available bandwidth. Based upon prior work and the analysis in Section 3, we conclude that such similarity exists in a number of important applications: large software updates (e.g., synchronizing a large collection of machines that may start with different versions of the software), transmitting large email attachments, and downloading multimedia content, among others. While our analysis focused on the latter category, we stress that there is nothing in SET that is specific to these file types. We emphasized them in our analysis because of their prevalence on file-sharing networks and because prior studies of similarity did not examine them.

Limited upstream bandwidth appears common in a number of peer-to-peer settings. File transfers in existing networks *are* slow; whether this occurs due to senders explicitly limiting their rates, ISPs policing traffic, or limited upstream speeds does not change the fact that receivers are unable to download at full speed. Regardless of the *cause* of the bottlenecks, being able to draw from more sources increases the likelihood of finding a fast source for the chunks a receiver desires.

8 Related Work

This paper builds on a rich volume of work in several areas: detecting and exploiting similar documents, systems for efficient multi-source bulk file transfer, and peer-to-peer lookup systems.

Similarity detection via "shingling" is a popular approach to creating metrics for the similarity between two or more documents, for purposes of analysis [17], clustering [4] or duplicate elimination [11]. Shingling runs a small sliding window (often 50 bytes) along a document, typically character by character or word-by-word, computing a checksum or hash at each step. Shingling then computes the final fingerprint of a document by deterministically sampling from those hashes. Handprinting differs in goal and analysis more than in mechanism: it is designed to detect exploitable similarity, not document resemblance, and we provide an analysis of its effectiveness when used as the basis for a distributed lookup mechanism. Denehy and Hu also study a similar "sliding-block" technique for filesystem data, finding that more than 32% of the data in their email corpus were duplicates [10].

Several distributed filesystems use chunk-level similarity to reduce duplicate transfers or storage requirements, including the Low Bandwidth File System [19], Casper [30], and the Pastiche [8] and Venti [25] backup systems. Of these systems, we call particular attention to Pastiche, in which nodes try to back up their data to a "backup buddy" that has substantial overlap in their filesystem contents. Pastiche accomplishes this in a local setting by having nodes send a random subset of their hashes to another node to determine similarity. We believe it would be interesting to explore whether handprints could help scale the Pastiche approach to larger clusters of machines. The work in [24] presents an analysis of chunk level similarity in different workloads.

Systems such as CFS [9] and Shark [3] exploit similarity at the chunk level. CFS stores and retrieves blocks from a DHT while Shark stores and retrieves chunk indexes from a DHT. In contrast to SET, they require $O(N)$ lookups—and perhaps more importantly, they require $O(N)$ state in the DHT. Shark improves on earlier DHT-based filesystems by taking advantage of locality for caching and fetching data.

SET builds upon much recent work in peer-to-peer file transfer [5, 7, 12, 14, 20, 21, 29]. SET draws techniques such as its end-game mode and its chunk selection heuristics from BitTorrent [7] and Bullet Prime [14]. A question for ongoing research is identifying which systems can use handprinting to find similar sources, and which cannot. Single-file mesh-based transfer systems such as Bullet Prime and SplitStream [5] are the next step, since they still transfer pieces of the original file. Extending SET to coding-based approaches such as Avalanche [12] seems more difficult, but poses an interesting challenge.

In our evaluation, we focused primarily on taking advantage of similarity when the number of sources were limited, or the available sources all had limited bandwidth. The focus of many prior per-file approaches (such as CoBlitz [20], and Bullet Prime [14]) was on making extremely efficient use of a larger mesh of nodes. Inasmuch as these protocols are used in a scenario where the overlay multicast mesh they create is capable of saturating the receivers' bandwidth, SET's techniques would provide no improvement. If, however, these systems were used in a situation where there was a nearby source of similar data with much better connectivity to the receiver, using similar sources would provide benefits similar to those we showed in Section 6.

SET depends on a global lookup table such as a Distributed Hash Table [26, 27]. SET is generally agnostic to the specifics of the lookup table; we believe it could run equally well on any lookup table that allows multiple values for a key and that allows some form of updates. Some retrieval systems such as ChunkCast [6] use the locality-aware structure of their global lookup tables (in their case, a DHT) to combine lookups with peer selection; we believe this approach is complementary to SET and handprinting, and that integrating the two is a promising avenue of future research.

9 Conclusion

This paper presented SET, a new approach to multi-source file transfers that obtains data chunks from sources of non-identical, but similar, files. We demonstrated that such similarity exists in active file sharing networks, even among popular multimedia content. SET employs a new technique called handprinting to locate these additional sources of exploitable similarity using only a constant number of lookups and a constant number of mappings per file. Evaluation results from a variety of network configurations showed that SET matches BitTorrent's performance without using similar sources, and exceeds BitTorrent's performance when exploiting similarity. SET's overhead is less than 0.5%.

We believe that handprinting strikes an attractive balance for multi-source transfers. It efficiently locates the sources of exploitable similarity that have the most chunks to contribute to a receiver, and it does so using only a small, constant number of lookups. For these reasons, we believe that this technique is an attractive one to use in any multi-source file transfer system.

Acknowledgments

We are particularly grateful to Niraj Tolia for extensive help with DOT and advice with content-addressable storage; to our NSDI shepherd and reviewers; to Nick Feamster, Mike Freedman, Phil Gibbons and Mary Lou Godbe for their helpful feedback. This work was supported in part by NSF CAREER award CNS-0546551.

References

[1] Gnutella. http://gnutella.wego.com, 2000.

[2] MLDonkey. `http://mldonkey.sourceforge.net/`.

[3] S. Annapureddy, M. J. Freedman, and D. Mazières. Shark: Scaling file servers via cooperative caching. In *Proc. 2nd USENIX NSDI*, Boston, MA, May 2005.

[4] A. Broder, S. Glassman, M. Manasse, and G. Zweig. Syntactic clustering of the web. In *Proceedings of the 6th International WWW Conference*, pages 1157–1166, Santa Clara, CA, Apr. 1997.

[5] M. Castro, P. Druschel, A.-M. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth content distribution in cooperative environments. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[6] B.-G. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In *Proc. 5th International Workshop on Peer-to-Peer Systems (IPTPS)*, Santa Barbara, CA, Feb. 2006.

[7] B. Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, Berkeley, CA, USA, June 2003.

[8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. 5th USENIX OSDI*, Boston, MA, Dec. 2002.

[9] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[10] T. E. Denehy and W. W. Hsu. Duplicate Management for Reference Data. Research Report RJ10305, IBM, Oct. 2003.

[11] F. Douglis and A. Iyengar. Application-specific delta-encoding via resemblance detection. In *Proceedings of the USENIX Annual Technical Conference*, San Antonio, Texas, June 2003.

[12] C. Gkantsidis and P. R. Rodriguez. Network coding for large scale content distribution. In *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005.

[13] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proc. 19th ACM Symposium on Operating Systems Principles (SOSP)*, Lake George, NY, Oct. 2003.

[14] D. Kostic, R. Braud, C. Killian, E. Vandekieft, J. W. Anderson, A. C. Snoeren, and A. Vahdat. Maintaining high bandwidth under dynamic network conditions. In *Proc. USENIX Annual Technical Conference*, Anaheim, CA, Apr. 2005.

[15] A. Legout, G. Urvoy-Keller, and P. Michiardi. Rarest first and choke algorithms are enough. In *Proc. ACM SIGCOMM Internet Measurement Conference*, Rio de Janiero, Brazil, Oct. 2006.

[16] J. Liang, R. Kumar, Y. Xi, and K. W. Ross. Pollution in p2p file sharing systems. In *Proc. IEEE INFOCOM*, Miami, FL, Mar. 2005.

[17] U. Manber. Finding similar files in a large file system. In *Proc. Winter USENIX Conference*, pages 1–10, San Francisco, CA, Jan. 1994.

[18] J. C. Mogul, Y. M. Chan, and T. Kelly. Design, implementation, and evaluation of duplicate transfer detection in HTTP. In *Proc. First Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, CA, Mar. 2004.

[19] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *Proc. 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, Oct. 2001.

[20] K. Park and V. Pai. Scale and Performance in the CoBlitz Large-File Distribution Service. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[21] K. Park and V. Pai. Deploying Large File Transfer on an HTTP Content Distribution Network. In *Proc. Workshop on Real, Large Distributed Systems (WORLDS)*, San Francisco, CA, Dec. 2004.

[22] K. Park, V. Pai, L. Peterson, and Z. Wang. CoDNS: Improving DNS performance and reliability via cooperative lookups. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[23] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proc. 1st ACM Workshop on Hot Topics in Networks (Hotnets-I)*, Princeton, NJ, Oct. 2002.

[24] C. Policroniades and I. Pratt. Alternatives for detecting redundancy in storage systems data. In *Proc. USENIX Annual Technical Conference*, Boston, MA, June 2004.

[25] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, CA, Jan. 2002.

[26] S. C. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A public DHT service and its uses. In *Proc. ACM SIGCOMM*, Philadelphia, PA, Aug. 2005.

[27] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. 18th IFIP/ACM International Conference on Distributed Systems Platforms*, Nov. 2001.

[28] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. Multimedia Computing and Networking (MMCN)*, Jan. 2002.

[29] R. Sherwood, R. Braud, and B. Bhattacharjee. Slurpie: A cooperative bulk data transfer protocol. In *Proc. IEEE INFOCOM*, Hong Kong, Mar. 2004.

[30] N. Tolia, M. Kozuch, M. Satyanarayanan, B. Karp, A. Perrig, and T. Bressoud. Opportunistic use of content addressable storage for distributed file systems. In *Proc. USENIX Annual Technical Conference*, pages 127–140, San Antonio, TX, June 2003.

[31] N. Tolia, M. Kaminsky, D. G. Andersen, and S. Patil. An architecture for Internet data transfer. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[32] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. 5th USENIX OSDI*, pages 255–270, Boston, MA, Dec. 2002.

[33] F. Zhou, L. Zhuang, B. Y. Zhao, L. Huang, A. Joseph, and J. Kubiatowicz. Approximate object location and spam filtering on peer-to-peer systems. In *Proc. ACM/IFIP/USENIX International Middleware Conference (Middleware 2003)*, 2003.

Cobra: Content-based Filtering and Aggregation of Blogs and RSS Feeds

Ian Rose, Rohan Murty, Peter Pietzuch, Jonathan Ledlie,
Mema Roussopoulos, and Matt Welsh
School of Engineering and Applied Sciences
Harvard University
{ianrose,rohan,prp,jonathan,mema,mdw}@eecs.harvard.edu

Abstract

Blogs and RSS feeds are becoming increasingly popular. The blogging site LiveJournal has over 11 million user accounts, and according to one report, over 1.6 million postings are made to blogs every day. The "Blogosphere" is a new hotbed of Internet-based media that represents a shift from mostly static content to dynamic, continuously-updated discussions. The problem is that finding and tracking blogs with interesting content is an extremely cumbersome process.

In this paper, we present Cobra (Content-Based RSS Aggregator), a system that crawls, filters, and aggregates vast numbers of RSS feeds, delivering to each user a personalized feed based on their interests. Cobra consists of a three-tiered network of *crawlers* that scan web feeds, *filters* that match crawled articles to user subscriptions, and *reflectors* that provide recently-matching articles on each subscription as an RSS feed, which can be browsed using a standard RSS reader. We present the design, implementation, and evaluation of Cobra in three settings: a dedicated cluster, the Emulab testbed, and on PlanetLab. We present a detailed performance study of the Cobra system, demonstrating that the system is able to scale well to support a large number of source feeds and users; that the mean update detection latency is low (bounded by the crawler rate); and that an offline service provisioning step combined with several performance optimizations are effective at reducing memory usage and network load.

1 Introduction

Weblogs, RSS feeds, and other sources of "live" Internet content have been undergoing a period of explosive growth. The popular blogging cite LiveJournal reports over 11 million accounts, just over 1 million of which are active over the last month [2]. Technorati, a blog tracking site, reports a total of 50 million blogs worldwide; this number is currently doubling every 6 months [38]. In July 2006, there were over 1.6 million blog postings every day. These numbers are staggering and suggest a significant shift in the nature of Web content from mostly static pages to continuously updated conversations.

The problem is that finding interesting content in this burgeoning blogosphere is extremely difficult. It is unclear that conventional Web search technology is well-suited to tracking and indexing such rapidly-changing content. Many users make use of RSS feeds, which in conjunction with an appropriate reader, allow users to receive rapid updates to sites of interest. However, existing RSS protocols require each client to periodically poll to receive new updates. In addition, a conventional RSS feed only covers an individual site, such as a blog. The current approach used by many users is to rely on RSS aggregators, such as SharpReader and FeedDemon, that collect stories from multiple sites along thematic lines (e.g., news or sports).

Our vision is to provide users with the ability to perform *content-based filtering and aggregation* across millions of Web feeds, obtaining a *personalized* feed containing only those articles that match the user's interests. Rather than requiring users to keep tabs on a multitude of interesting sites, a user would receive near-real-time updates on their personalized RSS feed when matching articles are posted. Indeed, a number of "blog search" sites have recently sprung up, including Feedster, Blogdigger, and Bloglines. However, due to their proprietary architecture, it is unclear how well these sites scale to handle large numbers of feeds, vast numbers of users, and maintain low latency for pushing matching articles to users. Conventional search engines, such as Google, have recently added support for searching blogs as well but also without any evaluation.

This paper describes Cobra (Content-Based RSS Aggregator), a distributed, scalable system that provides users with a personalized view of articles culled from potentially millions of RSS feeds. Cobra consists of a three-tiered network of *crawlers* that pull data from web feeds, *filters* that match articles against user subscriptions, and *reflectors* that serve matching articles on each subscription as an RSS feed, that can be browsed using a standard RSS reader. Each of the three tiers of the Cobra network is distributed over multiple hosts in the Internet, allowing network and computational load to be balanced, and permitting locality optimizations when placing services.

The core contributions in this paper are as follows. First, Cobra makes use of a novel offline *service provisioning* technique that determines the minimal amount of physical resources required to host a Cobra network capable of supporting a given number of source feeds and users. The technique determines the configuration of the network in terms of the number of crawlers, filters, and reflectors, and the interconnectivity between these services. The provisioner takes into account a number of characteristics including models of the bandwidth and memory requirements for each service and models of the feed content and query keyword distribution.

Our second contribution is a set of optimizations designed to improve scalability and performance of Cobra under heavy load. First, crawlers are designed to intelligently filter source feeds using a combination of HTTP header information, whole-document and per-article hashing. These optimizations result in a 99.8% reduction in bandwidth usage. Second, the filter service makes use of an efficient text matching algorithm [18, 29] allowing over 1 million subscriptions to match an incoming article in less than 20 milliseconds. Third, Cobra makes use of a novel approach for assigning source feeds to crawler instances to improve network locality. We perform network latency measurements using the King [24] method to assign feeds to crawlers, improving the latency for crawling operations and reducing overall network load.

The third contribution is a full-scale experimental evaluation of Cobra, using a cluster of machines at Harvard, on the Emulab network emulation testbed, and on PlanetLab. Our results are based on measurements of 102,446 RSS feeds retrieved from `i` `m` and up to 40 million emulated user queries. We present a detailed performance study of the Cobra system, demonstrating that the system is able to scale well to support a large number of source feeds and users; that the mean update detection latency is low (bounded by the crawler rate); and that our offline provisioning step combined with the various performance optimizations are effective at reducing overall memory usage and network load.

2 Related Work

Our design of Cobra is motivated by the rapid expansion of blogs and RSS feeds as a new source of real-time content on the Internet. Cobra is a form of *content-based publish-subscribe system* that is specifically designed to handle vast numbers of RSS feeds and a large user population. Here, we review previous work in pub-sub systems, both traditional and peer-to-peer designs. The database community has also developed systems for querying large numbers of real-time streams, some of which are relevant to Cobra.

Traditional Distributed Pub-Sub Systems: A number of distributed *topic-based* pub-sub systems have been proposed where subscribers register interest in a set of specific topics. Producers that generate content related to those topics publish the content on the corresponding *topic channels* [12, 22, 27, 5, 6, 7] to which the users are subscribed and users receive asynchronous updates via these channels. Such systems require publishers and subscribers to agree up front about the set of topics covered by each channel, and do not permit arbitrary topics to be defined based on a user's specific interests.

The alternative to the topic-based systems are *content-based* pub-sub systems [39, 37, 13, 40]. In these systems, subscribers describe content attributes of interest using an expressive query language and the system filters and matches content generated by the publishers to the subscribers' queries. Some systems support both topic-based and content-based subscriptions [32]. For a detailed survey of pub-sub middleware literature, see [31].

Cobra differentiates itself from other pub-sub systems in two ways. First, distributed content-based pub-sub systems such as Siena [13] leave it up to the network administrator to choose an appropriate overlay topology of filtering nodes. As a result, the selected topology and the number of filter nodes may or may not perform well with a given workload and distribution of publishers and subscribers in the network. By providing a separate provisioning component that outputs a custom-tailored topology of processing services, we ensure that Cobra can support a targeted work load. Our approach to pub-sub system provisioning is independent from our application domain of RSS filtering and could be used to provision a general-purpose pub-sub system like Siena, as long as appropriate processing and I/O models are added to the service provisioner.

Second, Cobra integrates directly with existing protocols for delivering real-time streams on the Web — namely, HTTP and RSS. Most other pub-sub systems such as Siena do not interoperate well with the current Web infrastructure, for example, requiring publishers to change the way they generate and serve content, and requiring subscribers to register interest using private subscription formats. In addition, the filtering model is targeted at structured data conforming to a well-known schema, in contrast to Cobra's text-based queries on (relatively unstructured) RSS-based web feeds.

Peer-to-Peer Overlay Networks: A number of content-delivery and event notification systems have been developed using peer-to-peer overlay networks, where the premise is that these systems are highly dynamic and can contain a large number of nodes exhibiting high rates of churn. As we explain in Section 3, we do not envision running Cobra in a peer-to-peer setting where nodes are contributed by volunteers, but instead assume the use of well-provisioned hosting centers, as is currently the norm for commercial Internet services. Nonetheless, it is worth describing some of the other design differences with these systems.

Corona [34] is a pub-sub system for the Web built on top of Pastry [35]. Users specify interest in specific URLs and updates are sent to users using instant message notifications. The main goal of Corona is to mitigate the *polling overhead* placed on monitored URLs, which is accomplished by spreading polling load among cooperating peers and amortizing the overhead of crawling across many users interested in the same URL. An informed algorithm determines the optimal assignment of polling tasks to peers to meet system-wide goals such as minimizing update detection time or minimizing load on content servers.

Corona is strictly concerned with allowing users to monitor an individual URL and focuses on the efficiency of the polling operation. Unlike Cobra, Corona does not permit an individual user to monitor a large number of web feeds simultaneously, nor specify content-based predicates on which content should be pushed to the user. Like Cobra, Corona can interoperate seamlessly with the current pull-based Web architecture.

A number of application-level multicast systems [36, 43] have been built using DHTs that construct an information dissemination tree with the multicast group members as leaf nodes. The resulting application-level multicast service is similar to the aforementioned topic-based pub-sub systems without content-based filtering. The multicast tree is formed by joining the routes from each subscriber node to a root node. In contrast, Cobra constructs an overlay topology using a service provisioning technique, taking into account the required resources to support a target number of source feeds and users.

Real-time Stream Querying: There has been much recent work from the database community on continuous querying of real-time data streams located at geographically dispersed data sources. These include Medusa [15], PIER [25], IrisNet [21], Borealis [14], and Stream-Based Overlay Networks [33]. These systems provide a general-purpose service for distributed querying of data streams, tend to assume a relational data model, and provide an elaborate set of operators to applications. Cobra, on the other hand, is specifically designed to filter and deliver relatively unstructured RSS feeds, provides a simple keyword-based query format to the user, and has models of the resource consumption of its crawler, filter, and reflector services used for provisioning.

Blog Search Engines: Recently, a number of "blog search engines" have come online, including Feedster, Blogdigger, Bloglines, IceRocket, and Technorati. Apart from Google and MSN's blog search services, most of these sites appear to be backed by small startup companies and little is known about how they operate. In particular, their ability to scale to large numbers of feeds and users, use network resources efficiently, and maintain low update latencies is unknown. In Section 4 we attempt to measure the update latency of several of these sites. As for MSN and Google, we expect these sites leverage the vast numbers of server resources distributed across many data centers to rapidly index updates to blogs. Although an academic research group cannot hope to garner these kinds of resources, by developing Cobra we hope to shed light on important design considerations and tradeoffs for this interesting application.

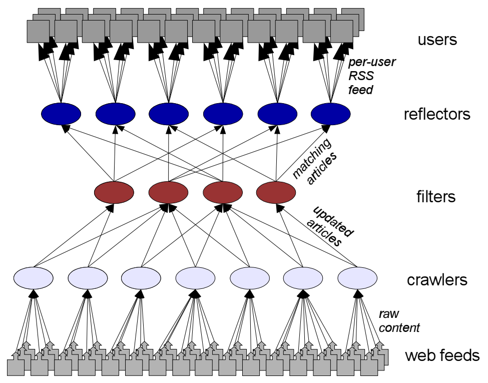


Figure 1: **The Cobra content-based RSS aggregation network.**

3 Cobra System Design

The overall architecture of Cobra is shown in Figure 1. Cobra consists of a three-tiered network of *crawlers*, *filters*, and *reflectors*. Crawlers are responsible for periodically crawling web feeds, such as blogs, news sites, and other RSS feeds, which we collectively call *source feeds*. A source feed consists of a series of *articles*. The number of articles provided by a source feed at any time depends on how it is configured; a typical blog or news feed will only report the most recent 10 articles or so. As described below, Cobra crawlers employ various techniques to reduce polling load by checking for updates in the source feeds in a lightweight manner.

Crawlers send new articles to the filters, which match the content of those articles against the set of user *subscriptions*, using a case-insensitive, index-based matching algorithm. Articles matching a given subscription are pushed to the appropriate reflector, which presents to the end user a personalized RSS feed that can be browsed using a standard RSS reader. The reflector caches the last k matching articles for the feed (where k is typically 10), requiring that the user poll the feed periodically to ensure that all matching articles will be detected. This behavior matches that of many existing RSS feeds that limit the number of articles included in the feed. Although the reflector must be polled by the user (as required by current

RSS standards), this polling traffic is far less than requiring users to poll many thousands of source feeds. Also, it is possible to replace or augment the reflector with push-based notification mechanisms using email, instant messaging, or SMS; we leave this to future work.

The Cobra architecture uses a simple congestion control scheme that applies backpressure when a service is unable to keep up with the incoming rate of data from upstream services. Each service maintains a 1MB data buffer for each upstream service. If an upstream service sends data faster than it can be processed, the data buffer will fill and any further send attempts will block until the downstream service can catch up, draining the buffer and allowing incoming data again. This ensures that the crawlers do not send new articles faster than the filters can process them, and likewise that the filters do not pass on articles faster than the reflectors can process them. The provisioner (detailed in section 3.5) takes this throttling behavior into account when verifying that each crawler will be able to finish crawling its entire list of feeds every 15 minutes (or whatever the *crawl-rate* is specified to be).

3.1 Crawler service

The crawler service takes in a list of source feeds (given as URLs) and periodically crawls the list to detect new articles. A naive crawler would periodically download the contents of each source feed and push all articles contained therein to the filters. However, this approach can consume a considerable amount of bandwidth, both for downloading the source data and sending updates to the filters. In a conventional usage of RSS, many users periodically polling a popular feed can have serious network impact [34]. Although Cobra amortizes the cost of crawling each feed across all users, the sheer number of feeds demands that we are careful about the amount of network bandwidth we consume.

The Cobra crawler includes a number of optimizations designed to reduce bandwidth usage. First, crawlers attempt to use the HTTP `a      i i` and `a` headers to check whether a feed has been updated since the last polling interval. Second, the crawler makes use of HTTP delta encoding for those feeds that support it.

When it is necessary to download the content for a feed (because its HTTP headers indicate it has changed, or if the server does not provide modification information), the crawler filters out articles that have been previously pushed to filters, reducing bandwidth requirements further and preventing users from seeing duplicate results. We make use of two techniques. First, a whole-document hash using Java's hashCode function is computed; if it matches the previous hash for this feed, the entire document is dropped. Second, each feed that has changed is broken up into its individual articles (or *entries*), which are henceforth processed individually. A hash is computed on each of the individual articles, and those matching a previously-hashed article are filtered out. As we show in Section 4, these techniques greatly reduce the amount of traffic between source feeds and crawlers and between crawlers and filters.

3.2 Filter service

The filter service receives updated articles from crawlers and matches those articles against a set of *subscriptions*. Each subscription is a tuple consisting of a *subscription ID*, *reflector ID*, and list of *keywords*. The subscription ID uniquely identifies the subscription and the reflector ID is the address of the corresponding reflector for that user. Subscription IDs are allocated by the reflectors when users inject subscriptions into the system. Each subscription has a list of keywords that may be related by either conjunctions (e.g. "law AND internet"), disjunctions (e.g. "copyright OR patent"), or a combination of both (e.g. "(law AND internet) OR (privacy AND internet)"). When an article is matched against a given subscription, each word of the subscription is marked either *true* or *false* based on whether it appears anywhere in the article; if the resulting boolean expression evaluates to *true* then the article is considered to have matched the subscription.

Given a high volume of traffic from crawlers and a large number of users, it is essential that the filter be able to match articles against subscriptions efficiently. Cobra uses the matching algorithm proposed by Fabret *et al.* [18, 29]. This algorithm operates in two phases. In the first phase, the filter service uses an index to determine the set of all words (across all subscriptions) that are matched by any article. This has the advantage that words that are mentioned in multiple subscriptions are only evaluated once. In the second phase, the filter determines the set of subscriptions in which all words have a match. This is accomplished by ordering subscriptions according to overlap and by ordering words within subscriptions according to selectivity, to test the most selective words first. If a word was not found in the first phase, all subscriptions that include that word can be discarded without further consideration. As a result, only a fraction of the subscriptions are considered if there is much overlap between them.

Due to its sub-linear complexity, the matching algorithm is extremely efficient: matching a single article against 1 million user subscriptions has a 90th percentile latency of just 10 ms (using data from real Web feeds and synthesized subscriptions, as discussed in Section 4). In contrast, a naive algorithm (using a linear search across the subscription word lists) requires more than 10 sec across the same 1 million subscriptions, a difference of four orders of magnitude.

3.3 Reflector service

The final component of the Cobra design is the reflector service, which receives matching articles from filters and reflects them as a personalized RSS feed for each user. In designing the reflector, several questions arose. First, should the filter service send the complete article body, a summary of the article, or only a link to the article? Clearly, this has implications for bandwidth usage. Second, how should filters inform each reflector of the set of matching subscriptions for each article? As the number of matching subscriptions increases, sending a list of subscription IDs could consume far more bandwidth than the article contents themselves.

In our initial design, for each matching article, the filter would send the reflector a summary consisting of the title, URL, and first 1 KB of the article body, along with a list of matching subscription IDs. This simplifies the reflector's design as it must simply link the received article summary to the personalized RSS feed of each of the matching subscription IDs. Article summaries are shared across subscriptions, meaning if one article matches multiple subscriptions, only one copy is kept in memory on the reflector.

However, with many active subscriptions, the user list could grow to be very large: with 100,000 matching subscriptions on an article and 32-bit subscription IDs, this translates into 400KB of overhead *per article* being sent to the reflectors. One alternative is to use a bloom filter to represent the set of matching users; we estimate that a 12KB filter could capture a list of 100,000 user IDs with a false positive rate of 0.08%. However, this would require the reflector to test each user ID against the filter on reception, involving a large amount of additional computation.

In our final design, the filter sends the complete article body to the reflector without a user list, and the reflector *re-runs* the matching algorithm against the list of active subscriptions it stores for the users it is serving. Since the matching algorithm is so efficient (taking 10ms for 1 million subscriptions), this appears to be the right trade-off between bandwidth consumption and CPU overhead. Instead of sending the complete article, we could instead send only the union of matching words across all matching subscriptions, which in the worst case reduces to sending the full text.

For each subscription, Cobra caches the last k matching articles, providing a personalized feed which the user can access using a standard RSS reader. The value of k must be chosen to bound memory usage while providing enough content that a user is satisfied with the "hits" using infrequent polling; typical RSS readers poll every 15-60 minutes [28]. In our current design, we set $k = 10$, a value that is typical for many popular RSS feeds (see Figure 5). Another approach might be to dynamically set the value of k based on the user's individual polling rate or the expected popularity of a given subscription. We leave these extensions to future work.

In the worst case, this model of user feeds leads to a memory usage of $k * subscriptions * 1KB$ (assuming articles are capped at 1KB of size). However, in practice the memory usage is generally much lower since articles can be shared across multiple subscriptions. In the event that memory does become scarce, a reflector will begin dropping the content of new articles that are received, saving to users' feeds only the articles' titles and URLs. This greatly slows the rate of memory consumption, but if memory continues to dwindle then reflectors will begin dropping *all* incoming articles (while logging a warning that this is happening). This process ensures a graceful degradation in service quality when required.

A user subscribes to Cobra by visiting a web site that allows the user to establish an account and submit subscription requests in the form of keywords. The web server coordinates with the reflectors and filters to *instantiate* a subscription, by performing two actions: (1) associating the user with a specific reflector node; and (2) injecting the subscription details into the reflector node and the filter node(s) that feed data into that reflector. The response to the user's subscription request is a URL for a private RSS feed hosted by the chosen reflector node. In our current prototype, reflector nodes are assigned randomly to users by the Web server, but a locality-aware mechanism such as Meridian [42] or OASIS [20] could easily be used instead.

3.4 Hosting model

Although "peer to peer," self-organizing systems based on shared resources contributed by volunteers are currently en vogue, we are not sure that this model is the best for provisioning and running a service like Cobra. Rather, we choose to exploit conventional approaches to distributed systems deployment, making use of well-provisioned hosting centers, which is the norm for commercial Internet services. Each of the three tiers of Cobra can be distributed over multiple hosts in the Internet, allowing computational and network load to be balanced across servers and hosting centers. Distribution also allows the placement of Cobra services to take advantage of improved locality when crawling blogs or pushing updates to users.

The use of a hosting center model allows us to make certain assumptions to simplify Cobra's design. First, we assume that physical resources in a hosting center can be dedicated to running Cobra services, or at least that hosting centers can provide adequate virtualization [4, 10] and resource containment [9] to provide this illusion.

Second, we assume that Cobra services can be replicated within a hosting center for increased reliability. Third, we assume that hosting centers are generally well-maintained and that catastrophic outages of an entire hosting center will be rare. Cobra can tolerate outages of entire hosting centers, albeit with reduced harvest and yield [19]. Finally, we assume that allocating resources to Cobra services and monitoring their performance at runtime can be performed centrally. These assumptions strongly influence our approach to service provisioning as we are less concerned with tolerating unexpected variations in CPU and network load and intermittent link and node failures, as is commonly seen on open experimental testbeds such as PlanetLab [30].

3.5 Service provisioning

As the number of source feeds and users grows, there is a significant challenge in how to provision the service in terms of computational horsepower and network bandwidth. Server and network resources cost money; additionally, a system may have limitations on the amount of physical resources available. Our goal is to determine the minimal amount of physical resources required to host a Cobra network capable of supporting a given number of source feeds and users. For this purpose, we make use of an offline *service provisioning* technique that determines the configuration of the Cobra network in terms of the number of crawlers, filters, and reflectors, as well as the interconnectivity between these services. Due to space constraints, we only provide an informal description of the service provisioning algorithm.

The provisioner takes as inputs the target number of source feeds and users, a model of the memory, CPU and bandwidth requirements for each service, as well as other parameters such as distribution of feed sizes and the per-user polling rate. The provisioner also takes as input a set of node constraints, consisting of limits on inbound and outbound bandwidth, maximum memory available to the JVM, and CPU processing power. Note that this last value is difficult to measure directly and thus we model it simply as a dimensionless parameter relative to the processing performance observed on Emulab's pc3000 machines [1]. For example, a CPU constraint of 0.75 implies that the provisioner should assume that nodes will process messages only 75% as fast as the pc3000s. The provisioner's output is a graph representing the topology of the Cobra network graph, including the number of feeds assigned to each crawler and the number of subscriptions assigned to each reflector and each filter.

The provisioner models each Cobra service as running on a separate physical host with independent memory, CPU and bandwidth constraints. This results in a conservative estimate of resource requirements as it does not permit multiple services within a hosting center to share resources (e.g., bandwidth). A more sophisticated algorithm could take such resource sharing into account.

The provisioner attempts to configure the network to meet the target number of source feeds and users while minimizing the number of services. The algorithm operates as follows. It starts with a simple 3-node topology with one crawler, one filter, and one reflector. In each iteration, the algorithm identifies any constraint violations in the current configuration, and greedily resolves them by *decomposing* services as described below. When no more violations exist, the algorithm terminates and reports success, or if a violation is found that cannot be resolved, the algorithm terminates and reports failure.

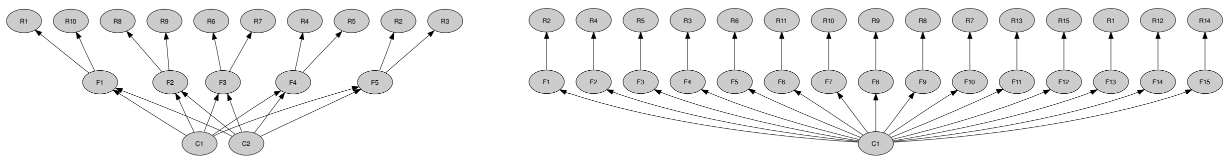
An *out-decomposition* resolves violations by replacing a single service with n replicas such that all incoming links from the original service are replicated across the replicas, whereas the outgoing links from the original services are load balanced across the replicas. An *in-decomposition* does the opposite: a single service is replaced by n replicas such that all outgoing links from the original service are replicated across the replicas, whereas the incoming links from the original services are load balanced across the replicas.

In resolving a violation on a service, the choice of decomposition type (in- or out-) depends both on the type of violation (in-bandwidth, out-bandwidth, CPU, or memory) and the type of service (crawler, filter or reflector). Figure 3 shows which decomposition is used in each situation.

When faced with multiple violations, the algorithm uses a few simple heuristics to choose the order in which to resolve them. Some violations have the potential to be resolved indirectly in the course of resolving other violations. For example, if a crawler service has both in-bandwidth and out-bandwidth violations, resolving the in-bandwidth violation is likely to also resolve the out-bandwidth violation (by reducing the number of feeds crawled, we also implicitly reduce the number of feed updates that are found and output by the crawler). Thus it is preferable in this case to resolve the in-bandwidth violation first as it may solve both violations with one decomposition. In general, when choosing which of multiple violations to resolve first, the algorithm will choose the violations with the least potential to be resolved indirectly, thus saving the violations with higher potential until as late as possible (in the hopes that they will "happen to be resolved" in the mean time).

Although this greedy approach might lead to local minima and may in fact fail to find a topology that satisfies the input constraints when such a configuration does exist, in practice the algorithm produces network topologies with a modest number of nodes to handle large loads. We chose this greedy iterative approach because it

[1] 3.0 GHz 64-bit Xeon processors



(a) Configuration for 4x CPU and 25 Mbps bandwidth

(b) Configuration for 1x CPU and 100 Mbps bandwidth

Figure 2: **Operation of the Cobra network provisioner.** *These figures show how provisioner results can vary for different constraint combinations; in both of these cases the network is provisioned for 800,000 feeds, 8 million subscriptions, and 1024 MB of memory, but the CPU and bandwidth constraints differ. (a) Shows the resulting configuration when the CPU constraint is 4x the default value (see text) and the bandwidth constraint is 25 Mbps. (b) Shows the resulting configuration when the CPU constraint is the default value (1x) and the bandwidth constraint is 100 Mbps. Compared to (a), this configuration requires half the number of crawlers, 50% more reflectors and three times as many filters (on account of the much greater processing needs).*

Service	Violation	Decomposition	Reason
Crawler	In-BW	In	Reduces the number of feeds crawled.
	Out-BW	In	Reduces the rate that updates are found and output to filters.
	CPU	None	Not modeled
	Memory	None	Not modeled
Filter	In-BW	In	Reduces the number of crawlers that send updates to each filter.
	Out-BW	In	Reduces the rate that articles are received, and thus also the rate that articles are output to reflectors.
	CPU	Out	Reduces the number of subscriptions that articles must match against.
	Memory	Out	Reduces the number of subscriptions that must be stored on the filter.
Reflector	In-BW	None	Not resolvable because reflectors must receive updates from all feeds (otherwise users will not receive all articles that match their subscription).
	Out-BW	Out	Reduces the subscriptions held by each reflector, which reduces the expected frequency of web queries by users.
	CPU	Out	Reduces the number of subscriptions that each incoming articles must be matched against and the number of article-queues that must be updated.
	Memory	Out	Reduces the number of subscriptions and article lists that must be stored.

Figure 3: **Provisioner choice of decomposition for each service/violation combination.**

was conceptually simple and easy to implement. Figure 2 shows two provisioner topologies produced for different input constraints.

3.6 Service instantiation and monitoring

The output of the provisioner is a *virtual graph* (see Figure 2) representing the number and connectivity of the services in the Cobra network. Of course, these services must be instantiated on physical hosts. A wide range of instantiation policies could be used, depending on the physical resources available. For example, a small startup might use a single hosting center for all of the services, while a larger company might distribute services across multiple hosting centers to achieve locality gains. Both approaches permit incremental scalability by growing the number of machines dedicated to the service.

The Cobra design is largely independent of the mechanism used for service instantiation. In our experiments described in Section 4, we use different strategies based on the nature of the testbed environment. In our dedicated cluster and Emulab experiments, services are mapped one-to-one with physical hosts in a round-robin fashion. In our PlanetLab experiments, services are distributed randomly to achieve good coverage in terms of locality gains for crawling and reflecting (described below). An alternate mechanism could make use of previous work on network-aware service placement to minimize bandwidth usage [8, 33].

After deployment, it is essential that the performance of the Cobra network be monitored to validate that it is meeting targets in terms of user-perceived latency as well as bandwidth and memory constraints. Also, as the user population and number of source feeds grow it will be essential to re-provision Cobra over time. We envision this process occurring over fairly coarse-grained time periods, such as once a month or quarter. Each Cobra node is instrumented to collect statistics on memory usage, CPU load, and inbound and outbound bandwidth consumption. These statistics can be collected periodically to ascertain whether re-provisioning is necessary.

3.7 Source feed mapping

Once crawler services have been instantiated, the final step in running the Cobra network is to assign source feeds to crawlers. In choosing this assignment, we are concerned not only with spreading load across multiple crawlers, but also reducing the total *network load* that the crawlers will induce on the network. A good way of reducing this load is to optimize the locality of crawlers and their corresponding source feeds. Apart from being good network citizens, improving locality also reduces the latency for crawling operations, thereby reducing the update detection latency as perceived by users. Because the crawlers use fairly aggressive timeouts (5 sec) to avoid stalling on slow feeds, reducing crawler-feed latency also increases the overall yield of a crawling cycle.

In Cobra, we assign source feeds to crawlers in a latency-aware fashion. One approach is to have each crawler measure the latency to all of the source feeds, and use this information to perform a coordinated allocation of the source feed list across the crawlers. Alternately, we could make use of network coordinate systems, such as Vivaldi [17], which greatly reduces ping load by mapping each node into a low-dimensional coordinate space, allowing an estimate of the latency between any two hosts to be measured as the Euclidean distance in the coordinate space. However, such schemes require end hosts to run the network coordinate software, which is not possible in the case of oblivious source feeds.

Instead, we perform an offline measurement of the latency between each of the source feeds and crawler nodes using King [24]. King estimates the latency between any two Internet hosts by performing an external measurement of the latency between their corresponding DNS servers; King has been reported to have a 75th percentile error of 20% of the true latency value. It is worth noting that many source feeds are hosted by the same IP address, so we achieve a significant reduction in the measurement overhead by limiting probes to those nodes with unique IP addresses. In our sample of 102,446 RSS feeds, there are only 591 unique IP addresses.

Given the latency matrix between feeds and crawlers, we perform assignment using a simple first-fit bin-packing algorithm. The algorithm iterates through each crawler C_j and calculates $i^\star = \arg\min l(F_i, C_j)$, where $l(\cdot)$ is the latency between F_i and C_j. $F_{i^\star}$ is then assigned to C_j. Given F feeds and C crawlers, we assign F/C feeds to each crawler (assuming $F > C$). We have considered assigning varying number of feeds to crawlers, for example, based on the posting activity of each feed, but have not yet implemented this technique.

Figure 4 shows an example of the source feed mapping from one of our experiments. To reduce clutter in the map we show only 3 crawlers (one in the US, one in Switzerland, and one in Japan) and the 5 nearest crawlers, according to estimated latency, for each. The mapping process is clearly effective at achieving good locality and naturally minimizes traffic over transoceanic links.

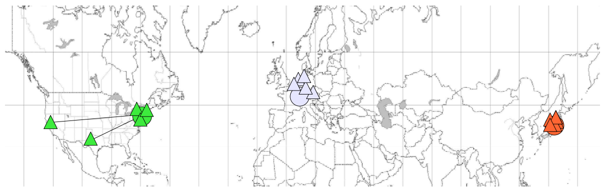


Figure 4: **An example of locality-aware source feed mapping.** *Three crawlers are shown as circles and the 5 nearest source feeds, according to estimated latency, are shown as triangles. Colors indicate the mapping from feeds to crawlers, which is also evident from the geographic layout.*

3.8 Implementation

Our prototype of Cobra is implemented in Java, and makes use of our substrate for *stream-based overlay networks* (SBONs) [33] for setting up and managing data flows between services. Note, however, that the placement of Cobra services onto physical hosts is determined statically, at instantiation time, rather than dynamically as described in our previous work [33]. A central *controller node* handles provisioning and instantiation of the Cobra network. The provisioner outputs a logical graph which is then instantiated on physical hosts using a (currently random) allocation of services to hosts. The instantiation mechanism depends on the specific deployment environment.

Our implementation of Cobra consists of 29178 lines of Java code in total. The crawler service is 2445 lines, the filter service is 1258 lines, and the reflector is 622 lines. The controller code is 377 lines, while the remaining 24476 consists of our underlying SBON substrate for managing the overlay network.

4 Evaluation

We have several goals in our evaluation of Cobra. First, we show that Cobra can scale well to handle a large number of source feeds and user subscriptions. Scalability is limited by service resource requirements (CPU and memory usage) as well as network bandwidth requirements. However, a modestly-sized Cobra network (Figure 2) can handle 8M users and 800,000 source feeds. Second, we show that Cobra offers low latencies for discovering matching articles and pushing those updates to users. The limiting factor for update latency is the rate at which source feeds can be crawled, as well as the user's own polling interval. We also present data comparing these update latencies with three existing blog search engines: Google Blog Search, Feedster, and Blogdigger.

We present results from experiments on three platforms: a local cluster, the Utah Emulab testbed [41], and PlanetLab. The local cluster allows us to measure service-level performance in a controlled setting, al-

	median	90th percentile
Size of feed (bytes)	7606	22890
Size of feed (articles)	10	17
Size of article (bytes)	768	2426
Size of article (words)	61	637

Figure 5: **Properties of Web feeds used in our study.**

though scalability is limited. Our Emulab results allow us to scale out to larger configurations. The PlanetLab experiments are intended to highlight the value of source feed clustering and the impact of improved locality.

We use a combination of real and synthesized web feeds to measure Cobra's performance. The real feeds consist of a list of 102,446 RSS feeds from syndic8.com, an RSS directory site. The properties of these feeds were studied in detail by Liu et al. in [28]. To scale up to larger numbers, we implemented an artificial *feed generator*. Each generated feed consists of 10 articles with words chosen randomly from a distribution of English words based on popularity rank from the Brown corpus [3]. Generated feed content changes dynamically with update intervals similar to those of real feeds, based on data from [28]. The feed generator is integrated into the crawler service and is enabled by a runtime flag.

Simulated user subscriptions are similarly generated with a keyword list consisting of the same distribution as that used to generate feeds. We exclude the top 29 most popular words, which are considered excessively general and would match essentially any article. (We assume that these words would normally be ignored by the subscription web portal when a user initially submits a subscription request.) The number of words in each query is chosen from a distribution based on a Yahoo study [11] of the number of words used in web searches; the median subscription length is 3 words with a maximum of 8. All simulated user subscriptions contain only conjunctions between words (no disjunctions). In Cobra, we expect that users will typically submit subscription requests with many keywords to ensure that the subscription is as specific as possible and does not return a large number of irrelevant articles. Given the large number of simulated users, we do not actively poll Cobra reflectors, but rather estimate the additional network load that this process would generate.

4.1 Properties of Web feeds

Liu et al. [28] present a detailed evaluation of the properties of RSS feeds, using the same list of 102,446 RSS feeds used in our study. Figure 5 summarizes the size of the feeds and individual articles observed in a typical crawl of this set of feeds between October 1–5, 2006. The median feed size is under 8 KB and the median number of articles per feed is 10.

Figure 6 shows a scatterplot of the size of each feed compared to its crawl time from a PlanetLab node running at Princeton. The figure shows a wide variation in the size and crawl time of each feed, with no clear relationship between the two. The large spike around size 8000 bytes represents a batch of 36,321 RSS feeds hosted by *topix.net*. It turns out these are not static feeds but dynamically-generated aggregation feeds across a wide range of topics, which explains the large variation in the crawl time.

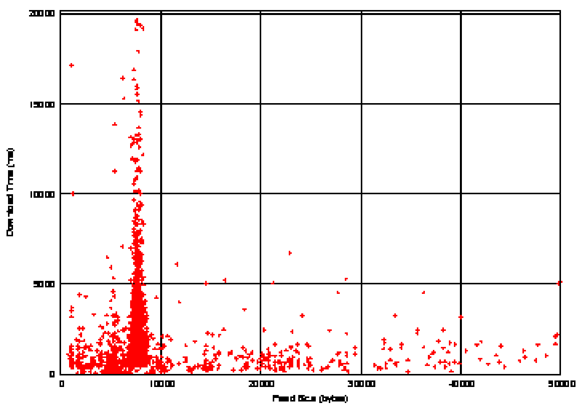


Figure 6: **Relationship between feed size and crawl time.**

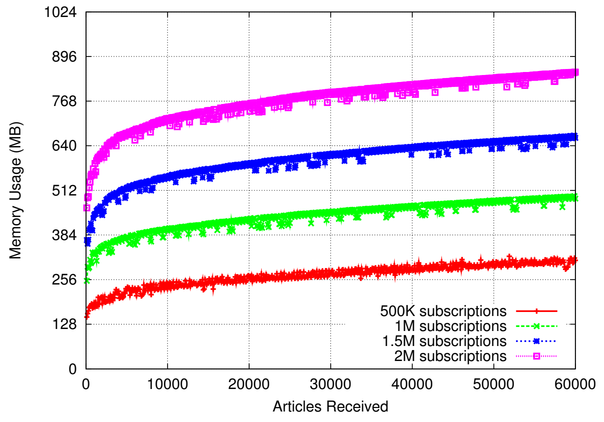


Figure 7: **Memory usage of the reflector service over time.** *The x-axis of this figure is the total number of articles received by the reflector. For context, we estimate that a set of 1 million feeds can be expected to produce an average of ∼48.5 updated articles every second, or ∼2910 each minute.*

4.2 Microbenchmarks

Our first set of experiments measure the performance of the individual Cobra services.

Memory usage

Figure 7 shows the memory usage of a single Reflector service as articles are received over time. In each case, the usage follows a logarithmic trend. However, the curves' obvious offsets make it clear that the number of subscriptions stored on each reflector strongly influences its memory usage. For example, with a half-million subscriptions, the memory usage reaches ∼310 MB after receiving 60,000 articles, whereas with 1 million subscriptions the memory usage reaches nearly 500 MB. This is not surprising; not only must reflectors store each actual subscription (for matching), but also each user's list of articles.

However, after the initial burst of article storage, the rate of memory consumption slows dramatically due to the cap (of $k = 10$) on each user's list of stored articles. This cap prevents users with particularly general subscriptions (that frequently match articles) from continually using up memory. Note that in this experiment no articles (or article contents) were dropped by the reflectors' scarce memory handling logic (as described in section 3.3). The only time that articles were dropped was when a user's list of stored articles exceeded the size cap.

This experiment assumes that articles are never expired from memory (except when a user's feed grows beyond length k). It is easy to envision an alternative design in which a user's article list is cleared whenever it is polled (by the user's RSS reader) from a reflector. Depending on the frequency of user polling, this may decrease overall memory usage on reflectors but an analysis of the precise benefits is left to future work.

In contrast, the memory usage of the crawler and filter services does not change as articles are processed. For crawlers, the memory usage while running is essentially constant since crawlers are unaffected by the number of subscriptions. For filters, the memory usage was found to vary linearly with the number of subscriptions (~0.16 MB per 1000 subscriptions held) and thus changes only when subscriptions are added or removed.

Crawler performance

Figure 8 shows the bandwidth reduction resulting from optimizations in the crawler to avoid crawling feeds that have not been updated. As the figure shows, using last-modified checks for reading data from feeds reduces the inbound bandwidth by 57%. The combination of techniques for avoiding pushing updates to the filters results in a 99.8% reduction in the bandwidth generated by the crawlers, a total of 2.2 KB/sec for 102,446 feeds. We note that none of the feeds in our study supported the use of HTTP delta encoding, so while this technique is implemented in Cobra it does not yield any additional bandwidth savings.

The use of locality-aware clustering should reduce the time to crawl a set of source feeds, as well as reduce overall network load. From our initial set of 102,446 feeds, we filtered out those that appeared to be down as well as feeds from two aggregator sites, *topix.net* and *izynews.de*, that together constituted 50,953 feeds. These two sites host a large number of dynamically-generated feeds that exhibit a wide variation in crawl times, making it difficult to differentiate network effects.

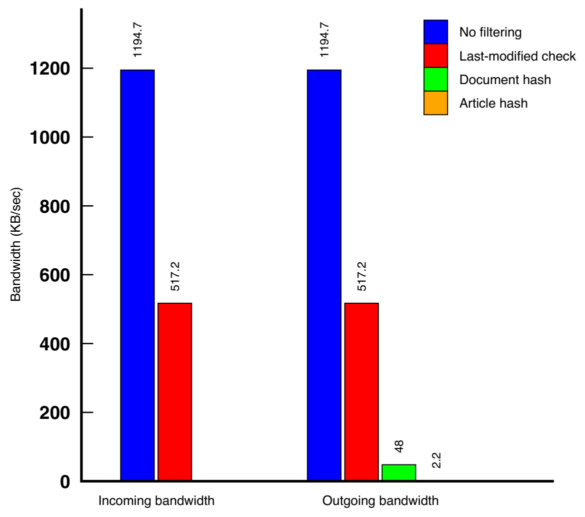


Figure 8: **Bandwidth reduction due to intelligent crawling.** *This graph shows the amount of data generated by the crawler using different techniques: (a) crawl all feeds; (b) filter based on last-modified header; (c) filter based on whole-document hash; and (d) filter based on per-article hashes.*

Figure 9 shows the time to crawl the remaining 34,092 RSS feeds distributed across 481 unique IP addresses. 11 crawlers were run on PlanetLab distributed across North America, Europe, and Asia. With locality aware mapping, the median crawl time per feed drops from 197 ms to 160 ms, a reduction of 18%.

Filter performance

Figure 10 shows the median time for the filter's matching algorithm to compare a single article against an increasing number of user subscriptions. The matching algorithm is very fast, requiring less than 20 ms to match an article of 2000 words against 1 million user subscriptions. Keep in mind that according to Figure 5 that the median article size is just 61 words, so in practice the matching time is much faster: we see a 90th percentile of just 2 ms per article against 1 million subscriptions. Of course, as the number of incoming articles increases, the overall matching time may become a performance bottleneck, although this process is readily distributed across multiple filters.

4.3 Scalability measurements

To demonstrate the scalability of Cobra with a large number of feeds and user subscriptions, we ran additional experiments using the Utah Emulab testbed. Here, we are interested in two key metrics: (1) The *bandwidth consumption* of each tier of the Cobra network, and (2) The

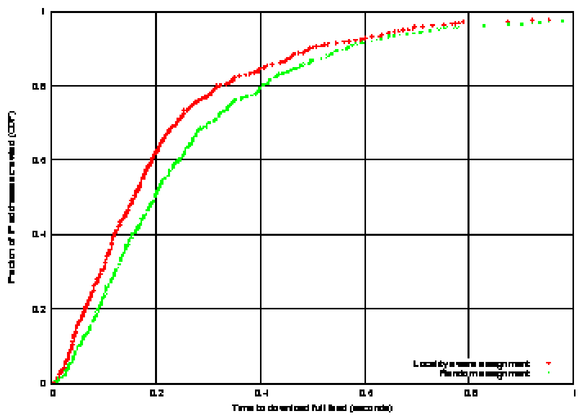


Figure 9: **Effect of locality-aware clustering.** *This is a CDF of the time to crawl 34092 RSS feeds across 481 separate IP addresses from 11 PlanetLab hosts, with and without the locality aware clustering.*

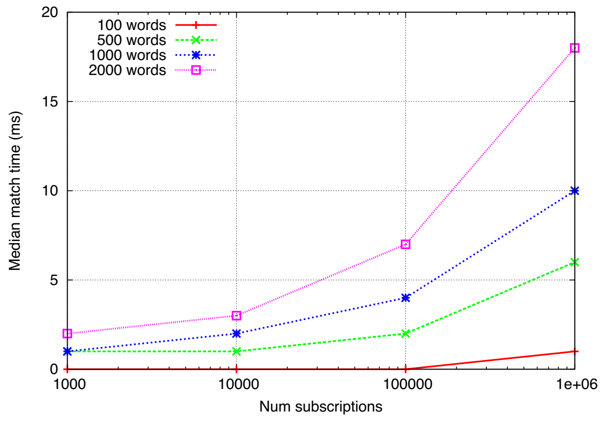


Figure 10: **Article match time versus number of subscriptions and number of words per article.** *The median time to match an article is a function of the number of subscriptions and the number of words per article.*

latency for an updated article from a source feed to propagate through the three tiers of the network. In total, we evaluated four different topologies, summarized in Figure 11.

Each topology was generated by the provisioner with a bandwidth constraint of 100 Mbps [2], a memory constraint of 1024 MB, and a CPU constraint of the default value (1x). In addition, we explicitly over-provisioned by 10% as a guard against bursty traffic or unanticipated bottlenecks when scaling up, but it appears that this was an largely unnecessary precaution. Each topology was run for four crawling intervals of 15 minutes each and the logs were checked at the end of every experiment to confirm that none of the reflectors dropped any articles (or article contents) to save memory (a mechanism invoked when available memory runs low, as discussed in section 3.3).

[2]We feel that the 100 Mbps bandwidth figure is not unreasonable; bandwidth measurements from PlanetLab indicate that the median inter-node bandwidth across the Internet is at least this large [26].

Subs	Feeds	Crawlers	Filters	Reflectors
10M	1M	1	28	28
20M	500,000	1	25	25
40M	250,000	1	28	28
1M	100,000	1	1	1

Figure 11: **Topologies used in scalability measurements.** *The last topology (100K feeds, 1M subscriptions) is meant to emulate a topology using the live set of 102,446 feeds.*

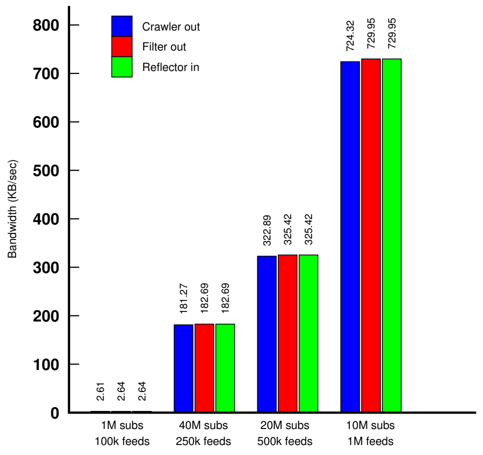


Figure 12: **Bandwidth consumption of each tier.** *The bandwidth of each tier is a function both of the number of feeds that are crawled and of the fan-out from each crawler to the filters.*

Figure 12 shows the total bandwidth consumption of each tier of the Cobra network for each of the four topologies evaluated. As the figure shows, total bandwidth consumption remains fairly low despite the large number of users and feeds, owing mainly to the effective use of intelligent crawling. Note that due to the relatively large number of subscriptions in each topology, the *selectivity* of the filter tier is nearly 1; every article will match some user subscription, so there is no noticeable reduction in bandwidth from the filter tier (the very slight increase in bandwidth is due to the addition of header fields to each article). One potential area for future work is finding ways to reduce the selectivity of the filter tier. If the filters' selectivity can be reduced, that will reduce not only the filters' bandwidth consumption, but also the number of reflectors needed to process and store the (fewer) articles sent from the filters. One way to lower filter selectivity may be to assign subscriptions to filters based on similarity (rather than the current random assignment); if all of the subscriptions on a filter tend towards a single, related set of topics, then more articles may fail to match any those subscriptions.

We are also interested in the *intra-network latency* for an updated article passing through the three tiers of the

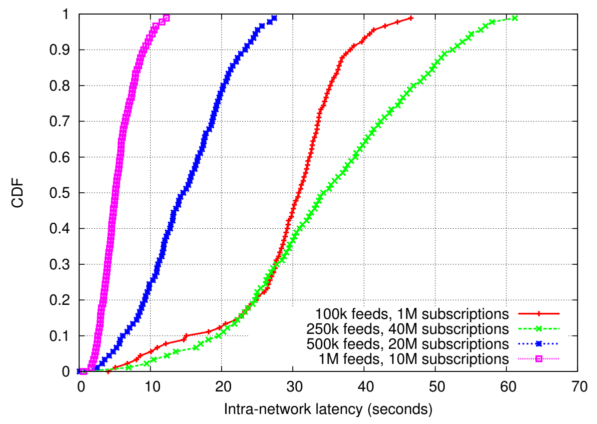


Figure 13: **CDF of intra-network latency for various topologies.** *This experiment shows that the intra-network latency is largely a factor of the processing load on filter and reflectors.*

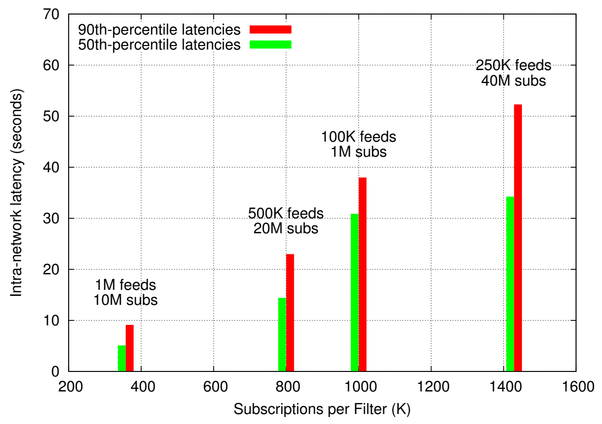


Figure 14: **Intra-network latency as a function of subscriptions per Filter.** *This figure shows the relationship between intra-network latency and the number of subscriptions stored on each Filter (note that in each of these topologies, the number of filters equals the number of reflectors, and thus the x-axis is equivalent to "Subscriptions per Reflector (K)").*

Cobra network. To gather this data, we instrumented the crawler, filter, and reflector to send a packet to a central logging host every time a given article was (1) generated by the crawler, (2) received at a filter, (2) matched by the filter, and (3) delivered to the reflector. Although network latency between the logging host and the Cobra nodes can affect these results, we believe these latencies to be small compared to the Cobra overhead.

Figure 13 shows a CDF of the latency for each of the four topologies. As the figure shows, the fastest update times were observed on the 1M feeds / 10M subs topology, with a median latency of 5.06 sec, whereas the slowest update times were exhibited by the 250K feeds / 40M subs topology, with a median latency of 34.22 sec. However, the relationship is not simply that intra-network latency increases with the number of users; the median latency of the 100K feeds / 1M subs topology was 30.81 sec - nearly as slow as the 250K feeds / 40M subs topology. Instead, latency appears more closely related to the number of subscriptions stored *per node* (rather than in total), as shown in Figure 14.

As mentioned at the end of section 3.2, nodes are able to throttle the rate at which they are passed data from other nodes. This is the primary source of intra-network latency; article updates detected by crawlers are delayed in reaching reflectors because of processing congestion on filters and/or reflectors. Since the time for a filter (or reflector) to process an article is related to the number of subscriptions that must be checked (see figure 10), topologies with larger numbers of subscriptions *per node* exhibit longer processing times, leading to rate-throttling of upstream services and thus larger intra-network latencies. Figure 14 shows a clear relationship between the number of subscriptions per node and the intra-network latencies. However, even in the worst of these cases, the latencies are still fairly low overall. As the system is scaled to handle more subscriptions and more users, Cobra will naturally load-balance across multiple hosts in each tier, keeping latencies low.

Note that the user's *perceived update latency* is bounded by the sum of the *intra-network latency* once an article is crawled by Cobra, and the *crawling interval*, that is, the rate at which source feeds are crawled. In our current system, we set the crawling interval to 15 minutes, which dominates the intra-network latencies shown in Figure 13. The intra-network latency is in effect the minimum latency that Cobra can support, if updates to feeds could be detected instantaneously.

4.4 Comparison to other search engines

Given the number of other blog search engines on the Internet, we were curious to determine how well Cobra's update latency compared to these sites. We created blogs on two popular blogging sites, LiveJournal and Blogger.com, and posted articles containing a sentence of several randomly-chosen words to each of these blogs.[3] We then searched for our blog postings on three sites: Feedster, Blogdigger, and Google Blog Search, polling each site at 5 sec intervals.

We created our blogs at least 24 hours prior to posting, to give the search engines enough time to index them. Neither Feedster or Blogdigger detected *any* of our postings to these blogs, even after a period of over four months (from the initial paper submission to the final camera-ready). We surmise that our blog was not indexed by these engines, or that our artificial postings were screened out by spam filters used by these sites.

Google Blog Search performed incredibly well, with a detection latency as low as 83 seconds. In two out

[3]An example posting was "certified venezuela gribble spork." Unsurprisingly, no extant blog entries matched a query for these terms.

of five cases, however, the latency was 87 minutes and 6.6 hours, respectively, suggesting that the performance may not be predictable. The low update latencies are likely the result of Google using a *ping service*, which receives updates from the blog site whenever a blog is updated [1]. The variability in update times could be due to crawler throttling: Google's blog indexing engine attempts to throttle its crawl rate to avoid overloading [16]. As part of future work, Cobra could be extended to provide support for a ping service and to tune the crawl rate on a per-site basis.

We also uncovered what appears to be a bug in Google's blog indexer: setting our unique search term as the title of the blog posting with no article body would cause Google's site to return a bogus results page (with no link to the matching blog), although it appears to have indexed the search term. Our latency figures ignore this bug, giving Google the benefit of the doubt although the correct result was not returned.

In contrast, Cobra's average update latency is a function of the crawler period, which we set at 15 minutes. With a larger number of crawler daemons operating in parallel, we believe that we could bring this interval down to match Google's performance. To our knowledge, there are no published details on how Google's blog search is implemented, such as whether it simply leverages Google's static web page indexer.

5 Conclusions and Future Work

We have presented Cobra, a system that offers real-time content-based search and aggregation on Web feeds. Cobra is designed to be incrementally scalable, as well as to make careful use of network resources through a combination of offline provisioning, intelligent crawling and content filtering, and network-aware clustering of services. Our prototype of Cobra scales well with modest resource requirements and exhibits low latencies for detecting and pushing updates to users.

Mining and searching the dynamically varying blogosphere offers many exciting directions for future research. We plan to host Cobra as a long-running service on a local cluster and perform a measurement study generated from real subscription activity. We expect our experience to inform our choice of parameters, such as how often to re-provision the system and how to set the number of articles cached for users (perhaps adaptively, depending on individual user activity). We also plan to investigate whether more sophisticated filtering techniques are desirable. Our current matching algorithm does not rank results by relevance, but rather only by date. Likewise, the algorithm is unconcerned with positional characteristics of matched keywords; as long as all keywords match an article, it is delivered to the user.

Unlike Web search engines, it is unclear what constitutes a good ranking function for search results on RSS feeds. For example, the link-based context such as that used by PageRank [23] may need to be modified to be relevant to Web feeds such as blog postings, which have few inbound links but often link to other (static) Web pages. Incorporating this contextual information is extremely challenging given the rapidly-changing nature of Web feeds.

Another open question is how to rapidly discover new Web feeds and include them into the crawling cycle. According to one report [38], over 176,000 blogs were created *every day* in July 2006. Finding new blogs on popular sites such as Blogger and LiveJournal may be easier than more generally across the Internet. While the crawler could collect lists of RSS and Atom URLs seen on crawled pages, incorporating these into the crawling process may require frequent rebalancing of crawler load. Finally, exploiting the wide distribution of update rates across Web feeds offers new opportunities for optimization. If the crawler services could learn which feeds are likely to be updated frequently, the crawling rate could be tuned on a per-feed basis.

References

[1] Google Blog Search FAQ. `http://www.google.com/help/blogsearch/about_pinging.html`.

[2] Livejournal. `http://www.livejournal.com`.

[3] Net dictionary index – brown corpus frequent word lising. `http://www.edict.com.hk/lexiconindex/`.

[4] Vmware esx server. `http://www.vmware.com/products/vi/esx/`.

[5] Js-javaspaces service specification, 2002. `http://www.jini.org/nonav/standards/davis/doc/specs/html/js-spec.html`.

[6] Tibco publish-subscribe, 2005. `http://www.tibcom.com`.

[7] Tspaces, 2005. `http://www.almaden.ibm.com/cs/TSpaces/`.

[8] Y. Ahmad and U. Çetintemel. Network-Aware Query Processing for Stream-based Applications. In *Proc. of VLDB'04*, Toronto, Canada, Aug. 2004.

[9] G. Banga, P. Druschel, and J. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. the Third OSDI (OSDI '99)*, February 1999.

[10] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *SOSP*, Oct. 2003.

[11] D. Bogatin. Yahoo Searches More Sophisticated and Specific. `http://blogs.zdnet.com/micro-markets/index.php?p=27`, May 18 2006.

[12] N. Carriero and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, Apr. 1989.

[13] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001.

[14] U. Centintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. ag S. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. Z. k. The Design of the Borealis Stream Processing Engine. In *Proc. of CIDR*, Asilomar, CA, Jan. 2005.

[15] M. Cherniack, H. Balakrishnan, M. Balazinska, et al. Scalable Distributed Stream Processing. In *Proc. of CIDR*, Asilomar, CA, Jan. 2003.

[16] M. Cutts. More webmaster console goodness. Matt Cutts: Gadgets, Google, and SEO (blog), `http://www.mattcutts.com/blog/more-webmaster-console-goodness/`, October 20 2006.

[17] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, Aug. 2004.

[18] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, and K. A. Ross. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe Systems. In *Proceedings of ACM SIGMOD*, 2001.

[19] A. Fox and E. A. Brewer. Harvest, yield and scalable tolerant systems. In *Proc. the 1999 Workshop on Hot Topics in Operating Systems*, Rio Rico, Arizona, March 1999.

[20] M. J. Freedman, K. Lakshminarayanan, and D. Mazieres. OASIS: Anycast for Any Service. In *Proc. USENIX/ACM NSDI*, San Jose, CA, 2006.

[21] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan. IrisNet: An Architecture for a World-Wide Sensor Web. *IEEE Pervasive Computing*, 2(4), Oct. 2003.

[22] B. Glade, K. Birman, R. Cooper, and R. van Renesse. Light-Weight Process Groups in the ISIS System. *Distributed Systems Engineering*, 1(1):29–36, 1993.

[23] S. Grin and L. Page. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *WWW7*, 1998.

[24] K. Gummadi, S. Saroiu, and S. D. Gribble. King: Estimating Latency between Arbitrary Internet End Hosts. In *Second Usenix/ACM SIGCOMM Internet Measurement Workshop*, Nov. 2002.

[25] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. tt Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of VLDB*, Berlin, Germany, Sept. 2003.

[26] S.-J. Lee, P. Sharma, S. Banerjee, S. Basu, and R. Fonseca. Measuring Bandwidth between PlanetLab Nodes. In *Proc. Passive and Active Measurement Workshop*, Boston, MA, March 2005.

[27] L.F.Cabera, M. Jones, and M. Theimer. Herald: Achieving a Global Event Notification Service. In *Workshop on Hot Topics in Operating Systems*, 2001.

[28] H. Liu, V. Ramasubramanian, and E. Sirer. Client Behavior and Feed Characteristics of RSS, a Publish-Subscribe System for Web Micronews. In *Proc. of ACM Internet Measurement Conference*, Oct. 2005.

[29] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient Matching for Web-Based Publish-Subscribe Systems. In *Proceedings of the 7th International Conference on Cooperative Information Systems (CoopIS)*, 2000.

[30] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Mui. Experiences Building PlanetLab. *OSDI*, Nov. 2006.

[31] P. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, Feb. 2004.

[32] P. Pietzuch and J. M. Bacon. Hermes: A Distributed Event-Based Middelware Architecture. In *1st Workshop on Distributed Event-Based Systems*, jul 2002.

[33] P. Pietzuch, J. Ledlie, J. Shneidman, M. R. M. Welsh, and M. Seltzer. Network-Aware Operator Placement for Stream-Processing Systems. In *ICDE*, Apr. 2006.

[34] V. Ramasubramanian, R. Peterson, and G. Sirer. Corona: A High Performance Publish-Subscribe System for the World Wide Web. In *NSDI*, 2006.

[35] A. Rowstron and P. Druschel. Pastry: Scalable, Decentralized Object Location and Routing for Large-Scale Peer-to-Peer Systems. In *Middleware)*, Nov. 2001.

[36] A. Rowstron, A. Kermarrec, M. Castro, and P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *NGC*, 2001.

[37] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin. In *ProceedingsofAUUG2K*, 2000.

[38] D. Sifry. State of the Blogosphere, August 2006. `http://www.sifry.com/alerts/archives/000436.html`.

[39] R. Strom, G. Banavar, T. Chandra, M. Kaplan, and K. Miller. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proceedings of the International Symposium on Software Reliability Engineering*, 1998.

[40] R. van Renesse, K. Birman, and W. Vogels. A Robust and Scalable Technology for Distributed Systems Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems*, 21(2):164–206, 2003.

[41] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI*, Dec. 2002.

[42] B. Wong, A. Slivkins, and G. Sirer. Meridian: A Lightweight Network Location Service without Virtual Coordinates. In *SIGCOMM*, Aug. 2005.

[43] S. Zhuang, B. Zhao, A. Joseph, R. Katz, and J. Kubiatowicz. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-Area Data Dissemination. In *NOSSDAV*, June 2001.

Information Slicing: Anonymity Using Unreliable Overlays

Sachin Katti skatti@mit.edu
Jeff Cohen jscohen@mit.edu
Dina Katabi dk@mit.edu

ABSTRACT

This paper proposes a new approach to anonymous communication called information slicing. Typically, anonymizers use onion routing, where a message is encrypted in layers with the public keys of the nodes along the path. Instead, our approach scrambles the message, divides it into pieces, and sends the pieces along disjoint paths. We show that information slicing addresses message confidentiality as well as source and destination anonymity. Surprisingly, it does not need any public key cryptography. Further, our approach naturally addresses the problem of node failures. These characteristics make it a good fit for use over dynamic peer-to-peer overlays. We evaluate the anonymity of information slicing via analysis and simulations. Our prototype implementation on PlanetLab shows that it achieves higher throughput than onion routing and effectively copes with node churn.

1 INTRODUCTION

Suppose Alice wants to send a confidential and anonymous message to Bob. Bob, however, does not have a public key that Alice could use to encrypt her message. Further, Alice does not feel comfortable exposing her unencrypted message to her ISP or an anonymizer. Alice's dilemma might seem simple, but underlying it is the general issue of online privacy. How do we send anonymous and confidential messages, when most of us do not have public keys and the sender does not trust a third party?

Our objective is to leverage popular existing peer-to-peer overlays to send confidential and anonymous messages without public keys. We focus on practical low-delay anonymity for everyday applications, rather than perfect anonymity at all costs. Popular peer-to-peer overlays have thousands of nodes and much traffic [5], creating an ideal environment for hiding anonymous communications. The dynamic nature of their participants makes them hard to track, and their diverse constituency allows dividing trust among nodes that are unlikely to collude. Some prior work has envisioned using these overlays for anonymity [15, 24, 21, 23, 16, 27]. Current proposals, however, fall into two camps: either they do not address the high node churn in these environments and need all overlay nodes to have public keys [15, 24, 21, 16], or they address churn but need very expensive solutions such as group key management [31] or broadcast [27].

This paper presents *information slicing*, a single technique that provides source and destination anonymity and churn resilience, without using any public key cryptography. It can also provide message confidentiality as long as the attacker cannot snoop on all traffic going to the destination. These characteristics make it suitable for use over popular peer-to-peer overlays. For example, say Alice knows that Bob, like many of us, uses a popular file sharing overlay to download content, and the overlay software supports information slicing. Then Alice can send Bob a confidential anonymous message without any public keys and in a manner robust to node churn and failures.

To provide confidentiality, our technique employs a properly chosen coding scheme to randomize the message. It then divides the randomized message into pieces, and sends the pieces along node disjoint paths that meet only at the destination. As a result, an attacker that gets all but one of the pieces of the randomized message cannot decode and recover the original message. Only the destination receives *all* pieces and can decode the message.

Information slicing also provides anonymity without requiring the overlay nodes to have public keys. Typically, anonymizers use onion routing, which assumes the sender has the public keys of all the nodes in the overlay. Onion routing hides the correspondence between a source and destination by sending the route setup message through a chain of nodes, wrapped in layers of public key encryption, such that each node along the path knows only its previous and next hops. Instead, to create an anonymous path, we send to each intermediate node its routing information (i.e., its next hop) in a confidential message sliced over multiple disjoint paths. The technical challenge is to perform this process efficiently. To send a relay node the identity of its next hop along different paths, we need to tell each node along these paths about its own next hop anonymously. Performed naively, this needs an exponential number of disjoint paths, and thus an exponential number of nodes. To avoid exponential blow-up, we build efficient forwarding graphs that reuse the overlay nodes without leaking information.

Finally, information slicing naturally provides protection against node churn and failures. The standard approach to address node failures is to employ multiple paths and add redundancy to the data. The challenge however is to minimize the redundancy overhead for the same amount of resilience. Typically, communication channels use coding to address such a challenge. We show that the same codes that we use to send confidential messages can simultaneously provide resilience to churn and failures. We also boost robustness by using network coding, which minimizes redundancy while maximizing resilience to failures [18].

We show analytically and demonstrate experimentally that information slicing provides high anonymity and is resilient to node churn. We implement our protocol and evaluate its real-world performance in PlanetLab. Our experimental results show that information slicing provides higher throughput than onion routing. Further, it provides strong resilience to node churn, while using minimal redundancy.

2 Related Work

First generation anonymizers used a single intermediate node to relay traffic between senders and receivers [1, 6]. Users had to trust the anonymizing node, which knows the identities of the source and destination.

Most modern anonymizers are based on Chaum mixes [9] and its near realtime variant, onion routing [17]. The sender constructs a route by picking intermediate hops from a list of mixing nodes. The mixers may delay and re-order the packets depending on the traffic's timing constraints. The sender encrypts the IP address of each node along the path with the public key of its previous hop. This creates layers of encryption, like layers of an onion. Each node decrypts the packets, discovers its next hop and forwards the packet to the next hop and so on until the entire path is set up. Once the path is established, nodes exchange computationally efficient symmetric secret keys to transmit the actual data itself.

A few anonymizers rely on static and dedicated overlays [12, 4, 3, 2]. For example, Tor [12] is a widely used anonymous system based on onion routing. Tor's infrastructure contains a small set of distributed nodes. Admission to the Tor network is tightly controlled. Tor has a centralized trusted directory server that provides the users with IP addresses and public keys of all the nodes in the system.

Some proposals [31, 15, 24, 16] aim to build anonymity out of global peer-to-peer overlays. Most of these systems employ onion routing and use public key cryptography. Only one of them addresses churn explicitly [31]. For example, Tarzan [15] uses onion routing, assumes each overlay node has a public key, and distributes these keys to interested senders using a gossip protocol. Tarzan sets up tunnels along each path, which are rebuilt upon node failures or departures. MorphMix's design is fairly similar to Tarzan and differs only in the details of the tunnel setup procedure [24]. Herbivore [16] builds on DC-nets [9] to provide anonymity in large overlays. It divides nodes into cliques and requires shared secrets for nodes across cliques via either a PKI or offline key exchanges. Freenet [10] is a decentralized censorship-resistant peer-to-peer data storage facility, intended for anonymous publishing, not communication.

Similar to ours, some prior work does not use public key cryptography. In Crowds [23], each intermediate node flips a coin to decide whether to forward a packet to the destination or to a random node in the overlay. In contrast to our work, Crowds does not provide destination anonymity, and uses a centralized admission server to admit nodes into the overlay. AP3 [21] is based on the same random routing idea, and similarly does not provide destination anonymity. P^5 [27] achieves anonymity by broadcasting encrypted packets at a constant rate to all participants. When a node has no packets to send, it broadcasts noise, which is then propagated through the network in the same manner as data packets. In comparison, our system does not broadcast messages and thus has a lower overhead. Finally, Malkhi et al. propose a system based on Secure Multi-Party Communication, which does not require cryptography [20]. They do, however, require secure channels between all participants. Such a requirement is hard to achieve in a large global overlay where most of the participants do not know each other a priori, and one cannot distinguish between good and bad participants.

To the best of our knowledge, there is only one prior proposal for addressing churn in anonymizing overlays. Cashmere [31] tackles churn by using a multicast group at each hop instead of a single node. Any node in the multicast group can forward the message. Cashmere assumes a trusted public key infrastructure (PKI) that assigns the same key to all nodes in each multicast group. Hence, Cashmere needs group key management and key redistribution, whenever group membership changes, which happens often in dynamic peer-to-peer overlays.

Finally, our information slicing idea is related to the theoretical work on secure communication [13, 29]. This work bounds the adversarial strength under which perfectly secure communication is possible. Our work on the other hand considers the problem of anonymous, confidential, and resilient communication. We provide stronger resilience to churn, a system implementation and evaluation of the performance of our protocol.

Some of the coding techniques used in our work are related to secret sharing [26]. A secret-sharing scheme is a method for distributing a secret among a group of participants, each of which is allotted a *share* of the secret. The secret can only be reconstructed when the shares are combined together, individual shares are of no use on their own. Our work, however, is significantly different from prior work on secret sharing; we focus on building a practical anonymizing overlay. Furthermore, our ideas about node reuse, the graph construction algorithm, and churn resilience are all different from secret sharing.

3 Model & Assumptions

(a) Goals: This paper aims to hide the source and destination identities, as well as the message content, from both external adversaries and the relay nodes. Further, the destination also does not know the identity of the actual source. Said differently, we are interested in the same type of anonymity exhibited in onion routing, where a relay node cannot identify the source or the destination, or decipher the content of the message; all it knows are its previous and next hops.

We also want a system that is practical and simple to deploy in a dynamic and unmanaged peer-to-peer overlay. The design should deal effectively with node churn. It must not need a trusted third party or a public key infrastructure, and preferably should not use any public key cryptography. The system also should not impose a heavy load on individual overlay nodes or require them to provide much bandwidth.

(b) Threat model: We assume an adversary who can observe a fraction of network traffic, operate relay nodes of his own, and compromise some fraction of the relays. Further, the compromised relays may also collude among themselves. Like prior proposals for low-latency anonymous routing, we do not address a global attacker who can snoop on all links in the network [12, 21, 15, 31]. Such a global attacker is unlikely in practice. We also assume that the attacker cannot snoop on all paths leading to the destination. If this latter assumption is unsatisfied, i.e., the attacker can snoop on all of the destination's traffic, the attacker can decode the content of the message but cannot identify the source of the message.

(c) Assumptions: We assume the source has an uncompromised IP address to access the Internet, S. Additionally, we assume the source has access to one or more IP addresses from which she can send. These IPs, which we call pseudo-sources S', should not be on the same local network as S. We assume that the source has a shared key with each of the pseudo-sources and communicates with them over a secure channel.

We believe these assumptions are reasonable. Many people have Internet access at home and at work or school, and thus can use one of these addresses as the source and the rest as pseudo-sources. Even when the user has only one IP address, she is likely to have a spouse, a friend, or a parent whose IP address she can use. When none of that is available, the user can go to an Internet cafe and use her home address as the source and the cafe's IP as a pseudo-source.

Note that the pseudo-sources cannot identify the destination or decipher the content of the message. They can only tell that the source is sending an anonymous message. In our system, we assume that the source wants to keep the pseudo-sources anonymous because they are personally linked to her, i.e., we protect the anonymity of the pseudo-sources in the same way as we protect the anonymity of the source. We conservatively assume that if the anonymity of any one of them is compromised then the source anonymity is also compromised. Thus, in the rest of this paper, the anonymity of the source comprises the anonymity of all pseudo-sources.

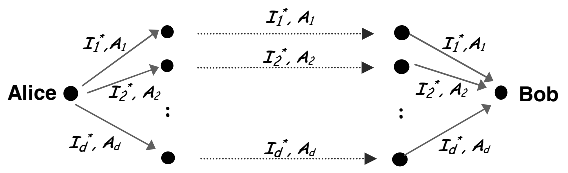


Figure 1—Alice wants to send a confidential message to Bob but does not know his key. Alice first multiplies the message $\vec{m}$ with a random matrix, A, then splits the resulting information, $\vec{I}^* = A\vec{m}$, into multiple pieces, $I_1^*, ..., I_d^*$. She sends each piece on a disjoint overlay path to Bob. Only Bob receives enough information bits to decode the original message as $\vec{m} = A^{-1}I^*$.

4 INFORMATION SLICING

The design of information slicing involves answering three questions:

- How do we send a confidential message without keys?
- How do we construct an anonymous overlay path? In particular, how do we hide the identities of the source and destination from the overlay nodes along the path and also hide the identity of the source from the destination?
- How do we make the protocol resilient to node churn?

We address each of these questions in the following sections, starting with message confidentiality.

4.1 Confidentiality Without Keys

Information slicing enables a source to send a confidential message to a destination without knowing the destination's key. Consider the scenario in Fig. 1. Alice wants to send the message "*Let's meet at 5pm*" to Bob. Alice divides the message into d pieces, e.g., m_1 ="*Let's meet*" and m_2 ="*at 5pm*" when $d = 2$, so that the original message can be recovered only when a node has access to all d pieces. We call this process *slicing the message*.

Sending a message slice in the clear is undesirable, as the slice may expose partial information to intermediate nodes along the path. For example, a node that sees m_1 ="*Let's meet*" knows that Alice and Bob are arranging for a meeting. Thus, Alice multiplies the message vector $\vec{m} = (m_1, ..., m_d$ with a *random but invertible* $d \times d$ matrix A and generates d slices which constitute a random version of the message:

$$\vec{I}^* = \begin{pmatrix} A_1 \\ \vdots \\ A_d \end{pmatrix} \vec{m} = A\vec{m}$$

Then, Alice picks d disjoint overlay paths to Bob. She sends on path i both the slice I_i^* and A_i, where A_i is row i of matrix A. An intermediate node sees only some random values I_i^* and A_i, and thus cannot decipher the content of the message. Once Bob receives all slices, he decodes the original message as:

$$\vec{m} = A^{-1}\vec{I}^*.$$

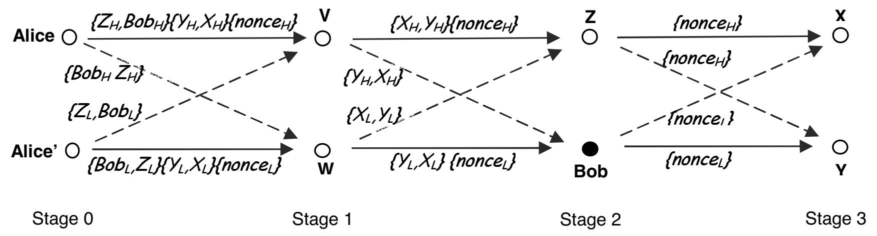


Figure 2—Alice wants to send an anonymous message to Bob without any public keys. Each node along the path needs to learn the IP addresses of its next hops in a confidential message, which is done by splitting each IP address and sending the pieces on disjoint paths. Alice has access to two machines *Alice* and *Alice′*. A message like $\{Z_l, Bob_l\}$ refers to the low-order words of the IDs of nodes *Z* and *Bob*, and *rand* refers to random bits.

4.2 Anonymous Routing Without Keys

Next, assume that Alice wants to send her message anonymously. How can Alice set up an anonymous path without keys? Each node along an anonymous path should know its previous hop and its next hop but nothing more. In onion routing, a node along the path learns its next hop from its previous hop – its parent. Though the parent delivers this information to its child, it cannot access it itself because the information is encrypted with the child's public key. In the absence of keys, the path cannot be included in the message, as that allows any intermediate node to learn the whole path from itself to the destination. We need an alternative method to tell a node about its next hop without revealing this information to other nodes, particularly parent nodes.

Our approach to anonymity without keys relies on a simple idea: *anonymity can be built out of confidentiality*. For anonymous communication, the source needs to send to every relay node along the path its routing information (i.e., its next hop) in a confidential message, accessible only to the intended relay. Information slicing enables a source to send such confidential messages without keys.

Using information slicing for anonymity, however, is challenging. To send a particular node the identity of its next hop along different anonymous paths, one needs to anonymously tell each node along these paths about its own next hop. This requires an exponential number of disjoint paths, and thus an exponential number of nodes. To avoid exponential blow-up, it is essential that the source constructs efficient forwarding graphs that reuse the overlay nodes without giving them too much information. The construction of such graphs in the general case is discussed in §*4.2.1*, but we first explain a simple example to give the reader an intuition about how the protocol works.

4.2.1 Example

Alice wants to send an anonymous message to Bob. Alice retrieves the DNS names of a few overlay nodes that she or her friends have used in the past to download music via a P2P file-sharing network. She can use DNS to retrieve the IP addresses of these overlay nodes. Alice does not know the public keys of the overlay nodes, or whether they have keys. She does, however, know that the software of the peer-to-peer overlay supports information slicing.

Alice has Internet at home and work, and hence has access to two IP addresses: *Alice* and *Alice′*. Alice arranges the overlay nodes into stages. Let's say she uses the graph in Fig. 2, which contains 3 stages (path length $L = 3$), each containing 2 nodes (split factor $d = 2$) (we will show how to pick appropriate values for L and d in §6). The 0^{th} stage is the source stage itself. Each node in this graph is connected to every node in its successive stage. Note that the destination node, i.e. Bob's node, is randomly assigned to one of the stages in the graph.

Alice needs to send to each relay node the IP addresses of its next hops, without revealing this information to other nodes. To do so, she splits each IP into two pieces and sends this information over two paths. Alice could have split each IP address to its most significant and least significant words. This, however, is undesirable as the most significant word may indicate the owner of the IP prefix. Instead Alice first transforms the IP addresses of the relay nodes by multiplying each address by an *invertible matrix* A of size $d \times d$ (i.e., 2×2). (For simplicity, assume that A is known to all nodes; in §4.3, we explain how the sender anonymously sends A to the relays on the graph.) Let Z_l and Z_h be the low and high words of the IP address of node Z; Alice splits the IP address as follows:

$$\begin{pmatrix} Z_L \\ Z_H \end{pmatrix} = A \begin{pmatrix} Z_l \\ Z_h \end{pmatrix}. \tag{1}$$

She sends Z_L and Z_H to node Z's parents, V and W, along two different paths.

Fig. 2 shows how messages are forwarded so that each node knows no more than its direct parents and children. Consider an intermediate node in the graph, say V. It receives the message "$\{Z_H, Bob_H\}\{X_H, Y_H\}\{rand_H\}$" from its first parent. It receives "$\{Z_L, Bob_L\}$" from its second parent. After receiving both messages, V can discover its children's IP addresses as follows:

$$\begin{pmatrix} Z_l & Bob_l \\ Z_h & Bob_h \end{pmatrix} = A^{-1} \begin{pmatrix} Z_L & Bob_L \\ Z_H & Bob_H \end{pmatrix} \tag{2}$$

But V cannot, however, identify the children of its children (i.e., the children of nodes Z and Bob) because it misses half the bits in these addresses, nor does it know the rest of the graph. Node V also does not know that Bob is the destination and Alice is the sender. From its perspective, Alice may have received the message from someone upstream, and Bob may be just another forwarder.

Var	Definition
d	Split factor, i.e., the number of slices a message is split to.
L	Path length, i.e., the number of relay stages along a path.
N	Number of nodes in the peer-to-peer network excluding the source stage.
f	Fraction of subverted nodes in the anonymizing network.

Table 1—Variables used in the paper.

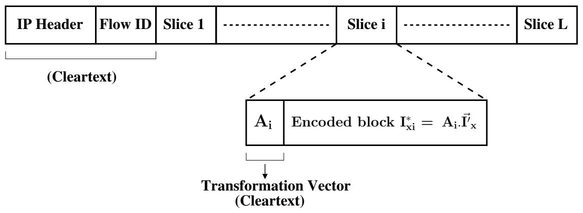


Figure 3—Packet Format. Each packet contains L information slices.

You might be wondering how the graph in Fig. 2 will be used to anonymously send data to Bob. Indeed, as it is, Bob does not even know he is the intended destination; but this is easy to fix. In addition to sending each node its next-hop IPs, Alice sends him: (1) a symmetric secret key, (2) and a flag indicating whether he is the destination. Similar to the next-hop IPs, the key and the flag of each node are split along disjoint paths, and thus inaccessible to other nodes. To send a confidential data message, Alice encrypts the data with the key she sent to Bob during the route setup phase, chops the data into d pieces, and forwards the pieces along the forwarding graph to Bob. Once Bob receives the d slices of the data, he can decode the encrypted data and invert the encryption using the key previously sent to him. No other node can decipher the data even if it gets all d slices.

4.3 Protocol Specification

This section rigorously describes our protocol. Our anonymous routing protocol delivers packets along a forwarding graph as explained in §*4.2.1*. The protocol has two phases. First, the source anonymously and confidentially informs each of the relay nodes on the graph of its forwarding information, i.e., it establishes the graph. Second, the source uses the forwarding graph to send data. If the source does not need to send much data, it is possible to collapse the two phases together and concatenate the data slices with the slices that build the graph. Before delving into the details of the protocol, we refer the reader to Table 1, which describes the variables used in the rest of the paper.

4.3.1 Per Node Information

Let x be one of the nodes in the forwarding graph. I_x is the routing information the source needs to *anonymously* deliver to node x. I_x consists of the following fields:

- *Nexthop IPs.* The IP addresses of node x's d children.
- *Nexthop flow-ids.* These are d 64-bit ids whose values are picked randomly by the source and are to be put in the clear in the packets going to the corresponding d next-hops. The source ensures that different nodes sending to the same next-hop put the same flow-id in the clear. This allows the next-hop to determine which packets belong to the same flow. The flow-id changes from one relay to another to prevent the attacker from detecting the path by matching flow-ids.
- *Receiver Flag.* This flag indicates whether the node is the intended destination.
- *Secret Key.* The source sends each node along the path a symmetric secret key that can be used to encrypt any further messages intended to this node.
- *Slice-Map.* This field describes which of the slices the relay receives go to which child (see §*4.3.4*).
- *Data-Map.* This field describes how the data packets flow down the graph (see §*4.3.7*).

4.3.2 Creating Information Slices

The source chops the node information I_x into d blocks of $\frac{|I_x|}{d}$ bits each and constructs a d length vector, $\vec{I_x}$. Further, it transforms $\vec{I_x}$ into coded *information slices* using a full rank $d \times d$ random matrix A as follows:[1]

$$\vec{I}^*_x = \begin{pmatrix} A_1 \\ \vdots \\ A_d \end{pmatrix} \vec{I_x} = A\vec{I_x} \tag{3}$$

The source concatenates each element in $\vec{I}^*_x$ with the row of the matrix A that created it (i.e., it concatenates I^*_{xi} with A_i). The result is what we call *an information slice*. The source delivers the d slices to node x along disjoint paths.

4.3.3 Packet Format

Fig. 3 shows the format of a packet used in our system. In addition to the IP header, a packet has a flow-id, which allows the node to identify packets from the same flow and decode them together. The packet also contains L slices. The first slice is always for the node that receives the packet. The other slices are for nodes downstream on the forwarding graph.

4.3.4 Constructing the Forwarding Graph

The source constructs a forwarding graph that routes the information slices to the respective nodes along vertex disjoint paths, as explained in Algorithm 1.

We demonstrate the algorithm by constructing such a graph in Fig. 4, where $L = 3$ and $d = 2$. The source starts with the 2 nodes in the last stage, X and Y. It assigns both the slices, I^*_{X1}, I^*_{X2} to X. The source then has to decide from whom node X will receive its slices. The source goes through the preceding stages, one by one, and distributes (I^*_{X1}, I^*_{X2}) among the 2 nodes at each stage. The distribution can be random as long as each node receives only one of the slices. The path taken by slice I^*_{X1} to reach X can be constructed by tracing it through the graph. For e.g., the slice

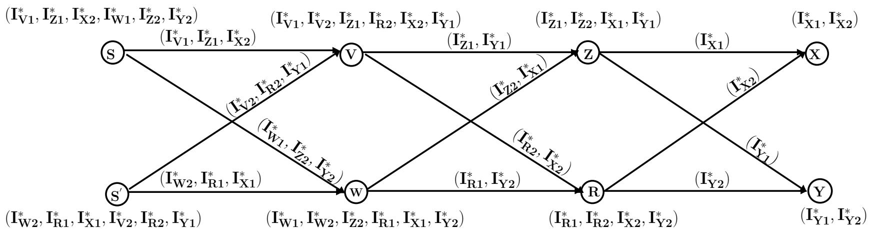


Figure 4—An example showing how to split information slices along disjoint paths. R is the destination, S and S' are the sources. The text on the arrows refers to the information slices traversing that edge. The text next to each node describes the slices collected by that node.

Algorithm 1 Information Slicing Algorithm

```
Pick $L \times d$ nodes randomly including the destination
Randomly organize the $L \times d$ nodes into $L$ stages of $d$ nodes each
for Stage $l = L$ to $l = 0$ do
    for Node $x$ in stage $l$ do
        Assign to node $x$ its own slices $I^*_{xk}, k \in (1, \ldots, d)$.
        for Stages $m = l - 1$ to $m = 1$ do
            Distribute slices $I^*_{xk}, k \in (1, \ldots, d)$ uniformly among the $d$ nodes in stage $m$, assigning one slice per node
        end for
    end for
    Connect every node in stage $l-1$ to every node in stage $l$ by a directed edge going towards $l$
    for every edge $e$ do
        Assign the slices that are present at both endpoints of Edge $e$ to the packet to be transmitted on $e$
    end for
end for
```

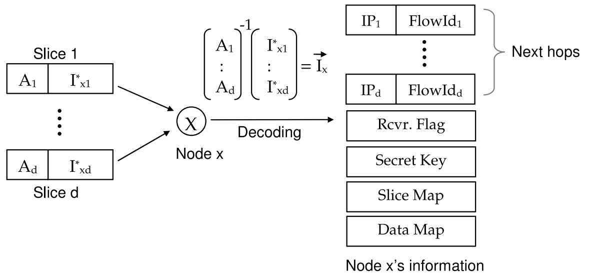


Figure 5—An example of how a node decodes its information from its incoming slices. It uses the *d* incoming slices and reconstructs the original information by inverting the matrix *A* and gets the IP addresses of its next-hops as well as the flows ids, its secret key, its slice-map, and its data-map.

I^*_{X1} traverses (S', W, Z, X), which is disjoint from the path taken by I^*_{X2}, i.e., (S, V, R, X). The source repeats the process for the slices of Y, and for the slices of every node in all the other stages.

4.3.5 Decoding the Information Slices

A node decodes its information from the d slices it receives from its parents, as shown in Fig. 5. The first slice in every packet that node x receives is for itself. It consists of one of the d-slices of x's information, I^*_{xi}, and the row of the transform matrix that helped create it, A_i. Node x constructs the vector $\vec{I}^*_x$ from the d slices it receives, and assembles a $d \times d$ matrix $A = [A_1; \ldots; A_d]$ from the d rows of the transform matrix sent in the slices. Then, node x computes its information vector, $\vec{I_x}$, as $\vec{I_x} = A^{-1}\vec{I}^*_x$.

4.3.6 How to Forward Information Slices

After the relay decodes its own information, it has to decide what to send to each one of its children. As is apparent from Figs. 2 and 4, a relay does not send the same information slices to all of its d children. The relay needs a map that tells it which of the information slices it received from its parents goes to which child, and in what order. This information is provided by the source in the slice-map.

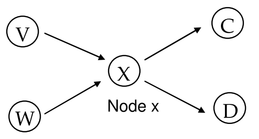
Fig. 6 shows an example slice-map. The slice-map is a $d \times L$ matrix that tells the relay how to construct the packets it sends to its children. For example, in Fig. 6, node x has received 2 packets from its parents. The number of slices in each packet is fixed to the path length $L = 4$. Note that x should first extract its own slice from each of the packets, which is the first slice in the packet, as explained in §*4.3.5*. The other three slices in each packet are to be forwarded downstream, as prescribed by x's slice-map. For example, the figure shows that the third slice in the packet received from V should be the first slice in the packet sent to node C. Entries labeled "rand" refer to padding slices with random bits that are inserted by the relay node to replace its own slices and maintain a constant packet size.

Additionally, all packets headed to a child node should contain the child node's slice as the first slice. The source constructs the slice maps of the parent nodes such that the packets meant for the child node always have the child node's slice as the first slice in the packet. Also the child node needs to be able to identify which d packets go together. The source arranges for all of the parent relays to use the same flow-id for packets going to the same child. The parent learns these flow-ids as part of its information, as shown in Fig. 5.

4.3.7 Data Transmission

Once the forwarding graph is established, the source can send anonymous data messages to the destination until it explicitly terminates the connection or the routing information times out. Also the destination can use a similar procedure to transmit to the source along the reverse path.

The source encrypts each data message with the key it sent to the destination node. Then it chops the data message into d pieces, converts them into d slices and multicasts the slices to the nodes in the first stage of the forwarding graph. Each relay node in the first stage receives all d data slices,



Child	Slice 1	Slice 2	Slice 3	Slice L
C	V3	rand	V4	W4
D	W2	V2	W3	rand

Slice-map for node x

Figure 6—An example showing the slice-map of node *X*, which has *V* and *W* as parents and *C* and *D* as children.

but they cannot multicast whatever they receive to the nodes in the next stage, since each child then will receive d^2 data slices leading to bandwidth overhead. On the other hand, if each node forwards a random slice to each of its children, then each child will get d data slices; but these slices may overlap and thus be useless. To solve the problem, the source sends each relay a data-map as part of its information. The data-map tells the node how to forward the data slices between each parent-child pair. The data map is very similar to the slice map shown in Fig. 6, except that instead of slice numbers the entries correspond to data packets. The source picks the entries in the data-map to ensure that each child gets all useful data slices, and no more. Each node in the graph including the destination therefore gets d slices, but since the data slices are encrypted using the destination's keys, only the destination can decrypt the data.

4.4 Resilience to Churn and Failures

Overlays with open membership suffer from churn because nodes join and leave frequently. Node churn causes data loss. The standard way to deal with loss is to add redundancy. The challenge, however, is to maximize the probability of recovery for the same amount of redundancy. Communication systems typically use *coding* to achieve this goal. Our design naturally extends the codes used to provide confidentiality to also provide resilience against churn and failures.

(a) Basic idea: Take a vector of d elements $\vec{m} = (m_1, \ldots, m_d)$ and multiply it by a random matrix A' of rank d and size $d' \times d$ where $d' > d$. The result will be a a vector of d' elements, $\vec{m}' = (m'_1, \ldots, m'_{d'})$; it is a redundant version of your original vector. What is interesting about this process is that it is possible to retrieve the original message from *any d* elements of $\vec{m}'$ and their corresponding rows in the matrix [18].

(b) Adding redundancy to graph establishment phase: Instead of slicing the per-node information into d independent pieces that are all necessary for decoding, we use $d' > d$ dependent slices. Replace Eq. 3 with:

$$\vec{I}^*_x = A'\vec{I}_x \tag{4}$$

where A' is a $d' \times d$ matrix with the property that any d rows of A' are linearly independent. The source picks d' disjoint paths to send the message. A node can recover its information from any d out of d' slices that it successfully receives.

(c) Adding redundancy to the data transfer phase: As mentioned earlier, the source encrypts the data with the symmetric key it sent to the destination during path establishment. The source then chops the encrypted message into d pieces, creating a message vector $\vec{m}$. Before it sends the message, however, it multiplies it by a random matrix A' of size $d' \times d$ and rank d, where $d' > d$. This creates d' data slices that the source sends along d' disjoint paths. The destination can recover the original information as long as it receives any d slices out of the d' data slices the source created.

4.4.1 Boosting Resilience to Churn Via Network Coding

The resilience scheme above is far from optimal. Consider an example where $d = 2$ and $d' = 3$, and assume that at some stage i along the path, one of the three relays fails. Its children in stage $i+1$ will receive two data slices instead of three. This is sufficient for recovering the original data. The problem, however, is that the redundancy is lost. Unless the redundancy is restored, downstream relays cannot recover from any additional failures.

We use network coding to solve the problem. Network coding allows intermediate nodes to code the data too. In our scheme, during the data transmission phase, a relay can easily restore the redundancy after its parent fails. To do so, the relay creates a linear combination of the slices it received, i.e., $m'_{new} = \sum p_i m'_i$, where p_i are random numbers. The relay also creates $A'_{new} = \sum p_i A'_i$, where p_i are the same numbers above. The new slice is the concatenation of A'_{new} and m'_{new} and can effectively replace the lost slice. Any relay that receives d or more slices can replace all lost redundancy. Thus, with a small amount of redundancy, we can survive many node failures because at each stage the nodes can re-generate the lost redundancy.

5 SECURITY ANALYSIS

Instead of standard key-based encryption, our scheme uses information slicing. To understand the level of confidentiality, i.e., the security obtained with such an approach, we estimate the amount of information a malicious node can glean from the messages it receives. We borrow the following definition from [8, 28].

Definition A function f is packet independent *pi*-secure if for all v and a uniformly distributed message block $\vec{x} = [x_1, x_2, \ldots, x_n]$ $Pr[x_i = v] = Pr[x_i = v | f(\vec{x})]$.

A *pi*-secure information slicing algorithm implies that to decrypt a message, an attacker needs to obtain all d information slices; partial information is equivalent to no information at all. The proof of the following lemma is in a technical report [19]:

LEMMA 5.1. *Information slicing is* pi-*secure.*

We note that there are many types of security, e.g., cryptographic security, pi-security, and information theoretic security. The strongest among these is information theoretic security. Information slicing can be made information theoretically secure, albeit with increased overhead. Instead of chopping the data into d parts and then coding them, we can combine each of the d parts with $d-1$ random parts. This will increase the space required d-fold, but provides extremely strong information-theoretic security.

6 Evaluation of Anonymity

The basic threat to anonymity in peer-to-peer overlays are attackers who compromise the overlay network. They can hack nodes, operate their own nodes, or eavesdrop on links to do traffic analysis. They can further collude to compromise anonymity. In this section we evaluate the anonymity of information slicing against such adversaries via simulations.

6.1 Anonymity Metric

The anonymity of a system is typically measured by its entropy [25, 11],[2] and is usually expressed in comparison with the maximum anonymity possible in such a system, i.e.:

$$Anonymity = \frac{H(x)}{H_{max}} = \frac{\sum_x -P(x)log(P(x))}{log(N)}, \qquad (5)$$

where N is the total number of nodes in the network, $P(x)$ is the probability of a node being the source/destination, and $H_{max} = log(N)$ is the maximum entropy that occurs when the attacker has no information. Anonymity is a number between 0 and 1. For example, the source is perfectly anonymous when it is equally likely to be any node in the network, in which case $P(x) = \frac{1}{N}$ and the $Anonymity = H(x)/H_{max} = 1$.

Note that $Anonymity = 0.5$ is quite high. It does not mean that the attacker knows the source or the destination with probability 0.5. Rather it means the attackers are still missing half the information necessary to discover the anonymous source or destination.

6.2 Simulation Environment

We would like to measure how the anonymity of the source and destination depends on the strength of the attackers. We simulate a scenario in which the attacker subverts a fraction f of the overlay nodes and the subverted nodes collude together. We assume that all attackers collude and consider them together as one powerful attacker. Note that this scenario subsumes attacks in which the attacker eavesdrops (i.e. does traffic analysis) on a fraction of the links because compromising a node is always a stronger attack than snooping on its input and output links. Further, this also subsumes "intersection" attacks in which attackers across multiple stages collude to compromise anonymity.

We assume that the source picks the relays randomly from the set of all nodes in the network, and that every node

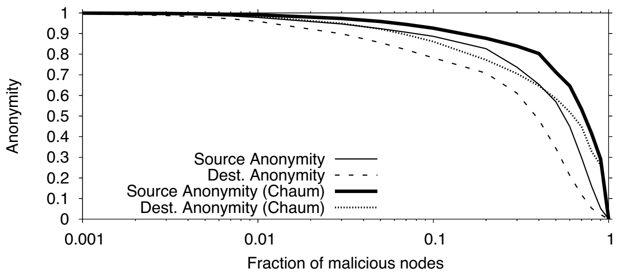


Figure 7—Source and destination anonymity as functions of the fraction of malicious nodes in the network ($N = 10000, L = 8, d = 3$). The anonymity obtained via information slicing is close to what Chaum mixes provide.

appears only once in the anonymity graph. These assumptions degrade anonymity, making our results conservative.

In each simulation, we randomly pick $f \times N$ nodes to be controlled by the attacker, where N is the number of overlay nodes. Then we pick $L \times d$ nodes randomly and arrange them into L stages of d nodes each. We randomly pick the destination out of the nodes on the graph. We identify the malicious nodes in the graph and analyze the part of the graph known to the attacker. Once we know the part of the graph known to the attacker, the anonymity for that *particular* scenario is computed. The details of how to compute source and destination anonymity for a particular simulation scenario are kept in a technical report [19]. Depending on the random assignment, the part of the graph known to the attacker will vary and so will the anonymity. Hence the simulation procedure is repeated 1000 times and the average anonymity is plotted.

6.3 Simulation Results

6.3.1 *Comparison with to Chaum mixes?*

In this section we evaluate the anonymity provided by information slicing and compare it to Chaum mixes. Consider attackers who compromise a fraction f of all nodes or links and collude together to discover the identities of the source and destination. Fig. 7 plots the anonymity of the source and destination as functions of the fraction of compromised nodes, for the case of $N = 10000, L = 8, d = 3$. When less than 20% of the nodes in the network are malicious, anonymity is very high and comparable to Chaum mixes, despite no keys. As the fraction of malicious nodes increases, the anonymity falls. But even when the attackers control half of the nodes, they are still missing half the information necessary to detect the source or destination. Destination anonymity drops faster with increased f because discovering the destination is possible if the attacker controls any stage upstream of the destination, while discovering the source requires the attacker to control stage 1, as we show in [19].

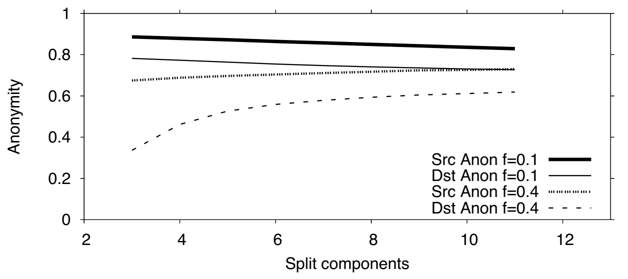


Figure 8—Source and destination anonymity as functions of the splitting factor ($N = 10000, L = 8$). For small f, increasing d decreases anonymity because it exposes more nodes to the attacker. For large f, the probability that attackers control an entire stage dominates, hence increasing d increases anonymity. Anonymity of 0.5 is still quite high since the attackers are missing half the information necessary to compromise the anonymity of the source and destination.

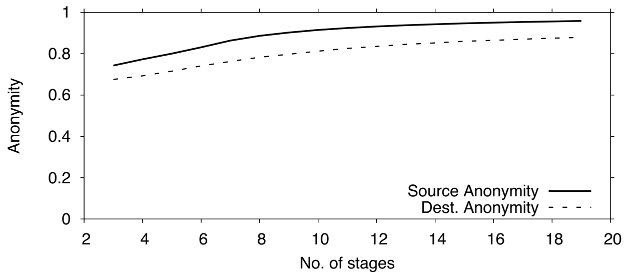


Figure 9—The anonymity of the source and destination increases with the path length ($N = 10000, d = 3, f = 0.1$).

6.3.2 Impact of Protocol Parameters

We evaluate how anonymity is affected by the parameters under the source's control: how many slices each piece of information is split into and the number of stages in the routing graph.

Fig. 8 plots source and destination anonymity as functions of the splitting factor d. When f is low, information leakage is due primarily to the malicious nodes knowing their neighbors on the graph. In this case, increasing d increases the exposure of non-malicious nodes to attackers which results in a slight loss of anonymity. When f is high, information leakage is mainly due to attackers being able to compromise entire stages. Hence, increasing d increases anonymity. However, even an anonymity as low as 0.5 is fairly strong; it means that the attacker is still missing half the bits necessary to identify the source/destination.

Fig. 9 plots the source and destination anonymity as functions of the path length L. Anonymity of both source and destination, increases with L. The attacker knows the source and destination have to be on the graph; thus, for moderate values of f, putting more nodes on the graph allows the communicators to hide among a larger crowd.

We also evaluate how anonymity and churn resilience trade off against each other. The theoretical analysis is in [19]. Fig. 10 plots the source and destination anonymity as functions of the redundancy added to combat churn. Redundancy is calculated as $(d' - d)/d$, and in the figure $d = 3$. As the added redundancy increases, it becomes more likely that the attacker compromises an entire stage of nodes. Hence

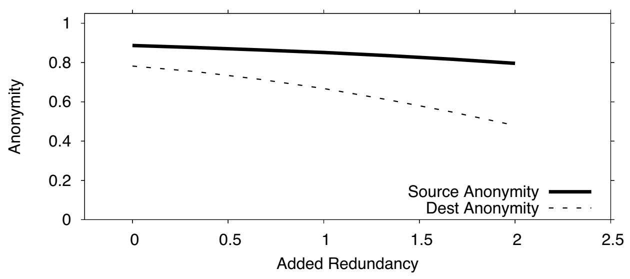


Figure 10—The anonymity of the destination decreases with the redundancy added (d = 3, L = 8, f = 0.1), source anonymity is not that adversely affected by redundancy since it is mainly a function of how many attackers are present in the graph.

destination anonymity decreases. Source anonymity is not much affected because it depends on whether specifically the first stage is compromised.

7 Evaluation of Performance

We evaluate the performance of information slicing via a prototype implementation run on the local and wide-area networks. Our wide-area experiments use 256 PlanetLab nodes, including nodes in North America, South America, Europe, and Asia. In each experiment, we pick a random subset of the above nodes depending on the size of the graph being set. We repeat each experiment 25 times by changing the randomly chosen subset of PlanetLab nodes and we take the average of the measured quantity. Our local-area experiments are performed on a 1 Gbps switched network with the nodes being 2.8 GHz Pentium boxes with 1 GB of RAM.

7.1 Implementation and Benchmarks

We have built a prototype of information slicing in Python. It includes two programs: an overlay daemon and a source utility. Each overlay node runs a multi-threaded daemon that listens on a special port. The daemon maintains a hash table keyed on the flow-id. For each anonymous flow, the table contains all the relevant forwarding information including the flow's next-hop IPs. When the daemon receives a packet, it forks a thread to process the packet and appropriately update the flow table. Additionally, the daemon periodically garbage collects the flow table to remove stale entries. The source utility program takes as input a list of willing overlay nodes, and a few configuration parameters such as the path length L, the number of parallel paths d', and the number of independent slices d.

The overhead of information slicing is low. We have performed benchmarks on a Celeron 800MHz machine with 256MB RAM connected to the local 1Gbps network. Coding and decoding require on average d finite-field multiplications per byte. Hence, the maximum achievable throughput is limited by how fast the multiplications can be accomplished. For $d = 5$, coding takes on average $60\mu s$ per 1500B packet, which limits the maximum output rate to 200Mbps. The memory footprint is determined by d, since we need

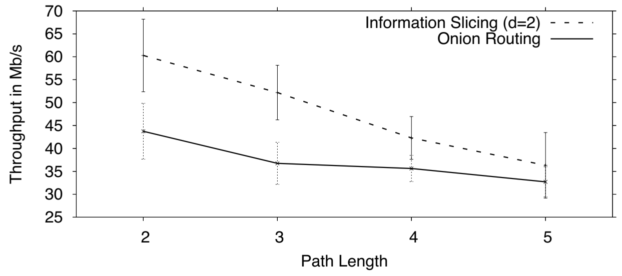


Figure 11—Comparison of the throughput of information slicing and onion routing on the local-area network. Information slicing achieves higher throughput due to the inherent parallelism involved in multiple paths.

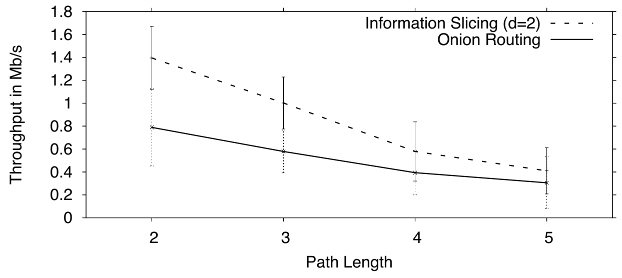


Figure 12—Comparison of the throughput of information slicing and onion routing on PlanetLab. Information slicing achieves higher throughput than onion routing.

the d packets to generate outgoing coded packets. Thus the memory consumed for packet storage is $d \times 1500B$ which is negligible.

7.2 Per-Flow Throughput

Fig. 11 and 12 show the throughput obtained when a transfer is run for 150 seconds using onion routing and our protocol for the local area network and PlanetLab respectively. The onion routing protocol uses computationally efficient symmetric session keys for the data transfer; public key cryptography is used only for the route setup. Both protocols use 1500 byte packets. On the local-area network (Fig. 11), our protocol can send at about 40-60 Mb/s. Our protocol achieves higher throughput than onion routing due to its parallelism. On PlanetLab (Fig. 12), the nodes are highly loaded, reducing the achievable throughput. Yet, the transfer achieves about 1 Mb/s, which is a good throughput for the wide-area network.

The overhead of information slicing in path setup is higher compared to onion routing. Specifically, since each message is split into d components, and each node outputs d packets in every round, the total number of packets between any two stages is d^2. For onion routing $d = 1$, whereas d can be varied in information slicing. But on the other hand, information slicing delivers higher throughput, since it used d parallel paths to deliver the data.

7.3 Scaling with the Number of Users

We examine how the throughput scales as the number of sources using the anonymizing overlay increases. Fig. 13

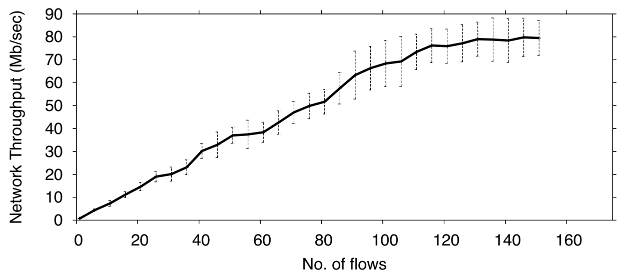


Figure 13—Network Throughput as a function of the number of flows using an overlay of 100 PlanetLab nodes. As load increases, the network throughput from information slicing scales almost linearly. At sufficiently high load, the network throughput levels off.

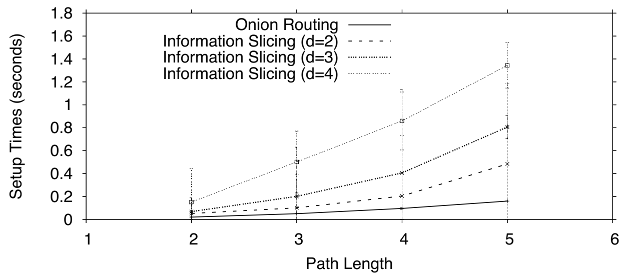


Figure 14—Average graph setup times on the local-area network as functions of path length L and splitting factor d. Increasing d means higher setup times, since each node has to wait for more packets before it can decode and forward its packet.

shows the *total* throughput as a function of the average load on the overlay nodes. The total throughput is the sum of the throughputs of the anonymous flows in the experiment. Load is quantified as the number of concurrent anonymous flows using the overlay. The experiment is performed on a set of 100 PlanetLab nodes that have long uptimes, so that churn does not affect the result (for churn results see 8). We set $d = 3$ and $L = 5$, hence each flow uses 15 nodes from the set of 100 nodes.

The figure shows that, as load increases, the total throughput scales linearly. At significantly high load, the throughput levels off and the overlay reaches saturation. The saturation threshold is a function of the used set of PlanetLab nodes and the loads imposed on these nodes by other users/experiments. Information slicing therefore scales well with the amount of load placed on the overlay up to moderate loads.

7.4 Route Setup Latency

Setup latency is measured end-to-end, from when the sender initiates the route establishment until the receiver sends back an ack (the ack is for measurement collection and not part of the protocol.) Our protocol allows the receiver to be randomly placed anywhere in the graph to obscure its identity. For purposes of our experiments, however, we place the receiver in the last stage of the graph, so that the measured setup times are the times to set up the entire graph, not just those stages up to the receiver's stage.

Fig. 14 plots the average graph setup times on the local-

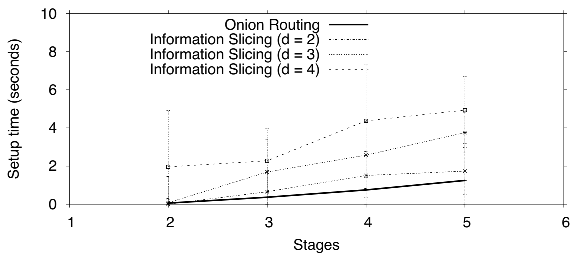


Figure 15—Average graph setup times on the wide-area network (PlanetLab) as functions of path length *L* and splitting factor *d*. Setup times are high since nodes have been picked all around the world and PlanetLab nodes were heavily loaded before the conference deadline.

area network. As one would expect, the setup time increases with increased path length L and splitting factor d. A large d affects the setup time because a relay has to wait to hear from all of its d parents and thus, the delay at each stage will be dominated by the slowest relay in that stage. In general however, the setup time is less than a couple of seconds. Furthermore, for $d = 2$, the setup time is a few hundred milliseconds.

We repeat the same experiments on PlanetLab to measure how much the conditions in the wide area network affect our setup times. Fig. 15 shows the average graph setup times in that environment. The setup times have increased beyond their values in the local-area network because of the larger RTT, but more importantly because PlanetLab nodes have a high CPU utilization leading up to the conference deadline. Despite this increase, the setup time is still within a few seconds.

8 Evaluation of Churn Resilience

Churn is an inescapable reality of dynamic peer-to-peer overlay networks. In §4.4 we presented a novel technique that recreates lost redundancy to combat churn. Here we evaluate its performance via analysis and an actual implementation. First we show analytically how coding helps us achieve high resilience with a small amount of added redundancy. Then we evaluate information slicing's churn resilience on PlanetLab and show that it can successfully cope with failures and make long anonymous transfers practical.

8.1 Analysis

We first show the efficiency of our coding approach compared to onion routing via analysis; but comparing it to standard onion routing would be unfair, as onion routing does not have any redundancy added and it would show very bad performance. Hence we compare it to a modified version of onion routing which has the same amount of redundancy as information slicing.

Imagine making onion routing resilient to failures by having the sender establish multiple onion-routing paths to the destination. The most efficient approach we can think of would allow the sender to add redundancy by using erasure codes (e.g., Reed-Solomon codes) over multiple onion routing paths. Assuming the number of paths is d', and the sender splits the message into d parts, she can then recover from any $d' - d$ path failures. We call this approach **onion routing with erasure codes**. Recall that in information slicing as well, the sender adds redundancy by increasing the number of paths $d' > d$, i.e., d slices of information are expanded to d' slices. But the key difference in information slicing is that relays *inside* the overlay network can regenerate lost redundancy.

To evaluate analytically, consider a message of S bytes. Suppose a sender has sent $S(1 + R)$ bytes, where R is the amount of redundancy in your transfer. R is also the overhead in the system; it limits the useful throughput. Now, let us compare the probability of successfully transferring the data under our scheme and onion routing with erasure codes when the same amount of redundancy is added. In particular, assume the path length is L, and that failures are independent and the probability of a node failing is p, the redundancy in both schemes is $R = \frac{d'-d}{d}$, hence $d' = (R + 1)d$.

Onion routing with erasure codes succeeds when there are at least d operational paths. A path is operational if none of the nodes on that path fail. Thus, the probability of a path success is $P(path\ succeeds) = (1 - p)^L$. The probability of the scheme succeeding is the probability of having at least d non-failing paths, i.e.,

$$P(success) \quad = \sum_{i=d}^{i=d'} \binom{d'}{i} (1-p)^{Li} \left(1-(1-p)^L\right)^{(d'-i)} \tag{6}$$

The information slicing approach, on the other hand, can tolerate $d' - d$ failures in each stage. The scheme succeeds if all stages succeed. A stage succeeds if at least d nodes in the stage do not fail, i.e.,

$$P(stage\ succeeds) = \sum_{i=d}^{i=d'} \binom{d'}{i} (1-p)^i p^{d'-i}$$

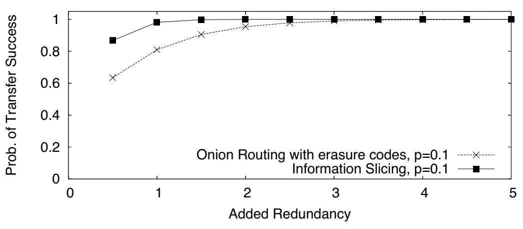
The slicing scheme succeeds if all stages succeed, i.e.:

$$P(success/slicing) = (P(stage\ succeeds))^L . \tag{7}$$

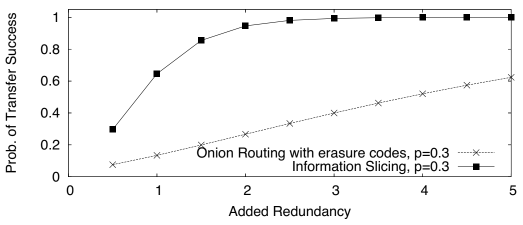
Fig. 16 illustrates the two success probabilities as a function of the amount of redundancy for $d = 2$, and $L = 5$. The probability of a node failure during the transfer is set to $p = 0.1$ in the top graph and $p = 0.3$ in the bottom graph. The figure shows that, for the same amount of overhead, the slicing approach has a substantially higher probability of successfully completing its transfer.

8.2 Resilience to Churn on PlanetLab

We complement the analytical evaluation with real experiments on a failure-prone overlay network, i.e., the PlanetLab network. We run our experiments with all nodes in our PlanetLab slice including the ones which are very failure prone. "Failure-prone" are nodes which are often inaccessible due to myriad reasons, either due to heavy CPU overload or network disconnectivity. These nodes have short "perceived" lifetimes of less than 20 minutes, and are extremely likely to fail during an experiment. The rationale



(a) Failure Prob is 0.1



(b) Failure Prob is 0.3

Figure 16—The probability of completing a transfer in information slicing and onion routing with redundancy as a function of the added redundancy. Figure shows that for the same level of redundancy, information slicing achieves much higher resilience to node failures ($L = 5, d = 2$).

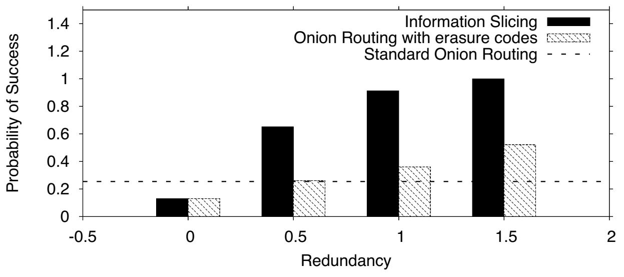


Figure 17—Resilience to node failures in PlanetLab for L = 5. Information slicing achieves very high resilience since it regenerates redundancy inside the network when a node fails.

for picking such nodes is that the sender usually has a list of overlay nodes, some of which are up and some are down. The sender cannot ping the nodes to identify the operational ones because this might jeopardize its anonymity.

We focus on the following question: “Given PlanetLab churn rate and failures, what is the probability of successfully completing a session that takes 30 minutes?” Given the throughput figures presented earlier, a 30-minute flow can transfer over 90 MB, which is typical of P2P file transfers. We compare information slicing with the modified version of onion routing which has redundancy added as described in the previous section.

Fig. 17 compares the probability of successfully finishing the transfer under our approach, standard onion routing (one path), and onion routing with redundancy added using erasure codes. As we saw in the previous section, for the same number of paths, onion routing with erasure codes has the same level of redundancy as our scheme. Redundancy is added by increasing the number of paths $d' > d$, in this case the added redundancy R is given by $(d'-d)/d$. The results are for $L = 5$ and $d = 2$. We vary the level of added redundancy by varying d', and measure the probability of successfully completing a session lasting 30 minutes.

The figure shows that with standard onion routing completing such a transfer is extremely unlikely. The probability of success increases with onion routing with erasure codes but stays relatively low. In contrast, with information slicing, adding a little amount of redundancy results in a very high success probability, making such relatively long anonymous transfers practical.

9 ROBUSTNESS TO ATTACKS

The biggest threat to peer-to-peer anonymizing overlays are from attackers who control nodes or can eavesdrop on links. Under conservative assumptions, i.e., even after assuming that an attacker who can eavesdrop on links leading to a node is as powerful as one who controls the node itself; we have shown that information slicing achieves anonymity comparable to Chaum mixes in §6. This section describes a few other attacks and how we address them. These attacks are fairly generic and apply to almost all anonymizers.

9.1 Limiting Malicious Nodes on the Graph

How does a sender choose relays for the anonymous graph it is setting up? One may be tempted to choose nodes completely at random from all available nodes; but an attacker could control large address spaces and increase the likelihood that the sender chooses colluding malicious nodes. To counter this attack, we leverage the structure of the IP address space. While an adversary can potentially control as many nodes as IP addresses to which he has access, these addresses are rarely scattered uniformly through the IP address space or through multiple autonomous systems (AS). IP addresses are divided into prefixes that are allocated to various networks worldwide. The prefixes appear in the inter-domain routing tables with their corresponding routes. These routing tables are *publicly available* from multiple vantage points [7]. It is realistic to assume that the attacker cannot compromise a large fraction of the inter-domain routing tables. Indeed if she can, then the attack has already jeopardized the Internet connectivity. By analyzing the publicly available routing tables, the sender can choose its relay nodes to be under different ASes potentially in different countries (e.g., Iran, US, China etc). This makes the above attack significantly more difficult because the attacker now needs to control many IP addresses belonging to many different ASes potentially spread around the world. Though this is possible, it is much more difficult. In the general case, picking overlay nodes that are independent and are guaranteed not to collude is a very difficult problem. Even if one knew the physical connectivity of the network, it is still not possible to guarantee non-collusion. Picking nodes based on their AS membership ensures that with high probability nodes are unlikely to collude.

9.2 Denial of Service Attack

It is always possible for a subverted relay to drop messages. It is also possible for a malicious source to try to consume the resources of the overlay nodes, denying other sources access to these resources. Overall, we believe that our approach neither increases nor decreases the vulnerability of an anonymizing overlay to denial of service; in comparison to onion routing, our approach allows the sender to store a small state (the per-node information) on the overlay nodes, but onion routing allows the sender to force the overlay nodes to do CPU-expensive public key cryptography.

In general, the best way to deal with denial of service attacks on anonymizing systems is to increase the size of the network. By allowing unmanaged peer-to-peer overlays with no trusted authority, our scheme has the potential to increase the size of these networks, thus increasing the resilience of the service.

9.3 Powerful Firewall

Consider a sender who lives under a repressive government that censors international online communications. The sender wants to anonymously communicate with an outside destination. To do so, it has to traverse the government's firewall. There are two cases. First the sender knows a pseudo-source outside the country. In this case, the sender splits the communication and securely tunnels a slice or more to outside pseudo-sources. The firewall, though it sees all slices, cannot reconstruct the message because some slices are encrypted. (Recall that a pseudo-source cannot read the message content or tell who the destination is.)

In the second case, the sender does not have access to a pseudo-source outside the firewall. In this case, the sender chooses some of the relays in some stage i to be outside the country and the rest inside –i.e., the firewall does not cut the graph at a single stage. For the firewall to be able to decipher the message, it needs to pick the right d^2 packets out of all packets in a particular interval (say 0.5s). These packets do not come from the same set of senders (because of the cross-stage cut) and the bits in these packets are hard to correlate. Furthermore, there are potentially billions of packets traversing the firewall during that interval. Picking the right d^2 packets therefore is a very difficult problem.

9.4 Traffic Analysis Attacks

There is always a tradeoff between robustness to attacks and increased overhead. Most solutions either send excessive amount of traffic or increase complexity making the system less usable. The right operation point usually depends on the application scenario. Our system focuses on providing practical low-delay anonymity for everyday applications rather than providing perfect anonymity at all costs. As mentioned in §3 we cannot protect against a global eavesdropper who can observe all traffic. Further if an attacker can snoop on all links leading to the destination, message confidentiality is compromised. But the attacker still cannot discover the identity of the sender.

Traffic analysis attacks become significantly harder to mount in a global overlay with thousands of nodes and a large amount of normal filesharing traffic. In predecessor attacks [30], the attacker forces frequent rebuilding of paths and tries to identify the sender and destination by identifying specific responders outside the overlay to which connections are made. For this attack, the attacker needs to observe all traffic across the global overlay which is unrealistic in practice. Murdoch et.al. [22] present an attack in which the attacker pings nodes in the overlay and identifies how the load offered by the adversary affects the delay of other anonymous communication streams at the source node. The attacker has to ping potentially thousands of nodes in the global overlay before he can observe any significant statistical change in the senders anonymous communication. Further the large amount of P2P filesharing traffic already present makes such attacks based on statistical analysis hard to mount in a global overlay network. We describe below some other specific traffic analysis attacks and how our system protects against them.

(a) Inserting a pattern in the data: Colluding attackers who are not in consecutive stages might try to track a connection by inserting a particular pattern in the packet and observing the path of the inserted pattern downstream. To prevent such attacks the sender makes the nodes along the path intelligently scramble each slice such that no pattern can percolate through the network. We will demonstrate the algorithm through a single slice K, which belongs to an intermediate node N in stage i. As we have seen before this slice passes through $i-1$ nodes before it reaches node N. Before transmitting the slice, the sender passes the slice through $i-1$ random transformations $T_1 * T_2 * \ldots * T_{i-1}$ successively. Now the sender has to ensure that when node N receives the slice, all of these random transformations have been removed, else the slice will be useless to node N. Therefore the sender confidentially sends each of the inverses of the $i-1$ random transformations applied above to the $i-1$ nodes which handle this slice. Each intermediate node applies one inverse transform to the slice K, hence by the time the slice reaches node N, the slice is through $i-1$ inverse transformations and is back to its original unmodified state. Node N can then decode and recover his own information.

The source repeats this process for all slices. This ensures that a slice is guaranteed to not look the same at any two links in the graph. Hence though the attacker might insert a particular pattern, the pattern will change in the immediate next stage without the attacker knowing how it changed. As a result, colluding non-consecutive attackers never see the same bit pattern, thereby nullifying the attack.

(b) End-to-End Time analysis attacks: The attacker may study the timing pattern in the traffic at various nodes in the overlay network in an attempt to identify where an anonymous flow enters and exits the overlay. Feamster et al [14] have shown that such an attack is possible in the Tor network. In particular, they report that Tor [12] has many nodes in the same AS. Hence, it is probable that the entry and exit nodes of an anonymous flow end up in the same AS. If this AS is adversarial, it can conduct timing analysis attacks to determine the sender and receiver. This attack becomes much harder in a large peer-to-peer overlay. In a large peer-to-peer network spread across the world, it is unlikely that a significant fraction of the nodes belong to the same AS. Furthermore, the node selection strategy outlined in §9.1 ensures that nodes are picked from different ASes, hence it is significantly hard for any single AS to mount a timing analysis attack.

(c) Resilience to packet size analysis: Looking at Fig. 4, the reader may think that the packet size decreases as it travels along the forwarding graph, and thus the attacker can analyze the position of a relay on the graph by observing the packet size. This is not the case, however. As explained in §*4.3.4*, each relay replaces its own information slices with random padding and shuffles the slices according to the sender's instructions sent in the slice-map. The map ensures that the packet size stays constant.

10 Conclusion

Anonymity spans a large design space; different anonymizers optimize for different objectives with corresponding trade-offs. Information slicing presents a unique point in this design space. It provides confidentiality, anonymity and churn resilience without public key cryptography but with the trade-off that the anonymity and confidentiality guarantees are slightly weaker than if we had public keys. We have analyzed the security and anonymity of the protocol and shown that it presents good guarantees against a variety of important attacks. We have also implemented the protocol, run it over PlanetLab, and shown that it is feasible and robust to node churn and failures. We believe information slicing simplifies the implementation and deployment of global peer-to-peer anonymizing networks, which is an important step towards scalable and practical online privacy.

11 Acknowledgments

We thank Jeremy Stribling, Vinod Vaikuntanathan, Hariharan Rahul, Maxwell Krohn, Frank Dabek and our shepherd Emin Gun Sirer for their comments on the paper. Cohen and Katabi are supported by the NSF Career Award CNS-0448287, whereas Katti is supported by NSF award CNS-0627021. The opinions and findings in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Anonymizer- Anonymous Web Surfing. http://www.anonymizer.com.
[2] Freedom Network. http://www.freedom.net/.
[3] JAP Anonymity and Privacy. http://anon.inf.tu-dresden.de/index_en.html/.
[4] MixMinion- Anonymous Remailer. http://www.mixminion.net.
[5] Peer-to-Peer in 2005. http://www.cachelogic.com/home/pages/research/p2p2005.php.
[6] Safeweb- Anonymous Web Surfing. http://www.safeweb.com.
[7] University of Oregon Route Views Project. http://www.routeviews.org/.
[8] J. Byers, M. C. Cheng, J. Considine, G. Itkis, and A. Yeung. Securing Bulk Content Almost for Free. *Journal of Computer Communications, Special Issue on Network Security*, pages 280–290, February 2006.
[9] D. L. Chaum. Untraceable electronic mail, return addresses and digital pseudonyms. *Commun. of the ACM*, pages 84–90, February 1981.
[10] I. Clarke. FreeNet. http://freenet.sourceforge.net/.
[11] C. Díaz, S. Seys, J. Claessens, and B. Preneel. Towards measuring anonymity. In *Proc. of PET 2002*, San Francisco, CA, USA.
[12] R. Dingledine, N. Mathewson, and P. Syverson. TOR: The second-generation onion router. In *Proc. of USENIX Security 2004*, San Diego, CA, USA.
[13] D. Dolev, C. Dwork, O. Waarts, and M. Yung. Perfectly secure message transmission. *Journal of the ACM*, 40(1), 1993.
[14] N. Feamster and R. Dingledine. Location diversity in anonymity networks. In *In Proc. of ACM WPES 2004.*
[15] M. J. Freedman and R. Morris. Tarzan: A peer-to-peer anonymizing network layer. In *Proc. of ACM CCS 2002*, Washington, DC, USA.
[16] S. Goel, M. Robson, M. Polte, and E. Sirer. Herbivore: A scalable and efficient protocol for anonymous communication. In *Proc. of the European SIGOPS Workshop 2004*, Leuven, Belgium.
[17] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding Routing Information. In *Proc. of Workshop on Information Hiding*, Cambridge, UK, 1996.
[18] T. Ho, M. Médard, J. Shi, M. Effros, and D. Karger. On randomized network coding. In *Proc. of Allerton Conference on Communication, Control, and Computing*, Monticello, IL, 2003.
[19] S. Katti, J. Cohen, and D. Katabi. Information slicing: Anonymity using unreliable overlays. Technical Report MIT-CSAIL-TR-2007-013, CSAIL, MIT, February 2007.
[20] D. Malkhi and E. Pavlov. Anonymity without cryptography. In *Financial Cryptography 2001*, Grand Cayman, British West Indies.
[21] A. Mislove, G. Oberoi, A. Post, C. Reis, P. Druschel, and D. Wallach. Ap3: A cooperative, decentralized service providing anonymous communication. In *Proc. of the ACM SIGOPS European Workshop, 2004*, Leuven, Belgium.
[22] S. Murdoch and G. Danezis. Low-cost traffic analysis for TOR. In *Proc. of IEEE Symposium on Security and Privacy 2005*, Oakland, CA, USA.
[23] M. Reiter and A. Rubin. Crowds: Anonymity for web transactions. *ACM Transactions on Information and System Security*, 1(1), June 1998.
[24] M. Rennhard and B. Plattner. Introducing MorphMix: Peer-to-Peer based Anonymous Internet Usage with Collusion Detection. In *Proc. of WPES 2002*, Washington, DC, USA.
[25] A. Serjantov and G. Danezis. Towards an information theoretic metric for anonymity. In *Proc. of PET 2002*, San Francisco, CA, USA.
[26] A. Shamir. How to share a secret. *Communications of the ACM*, pages 612–613, November 1979.
[27] R. Sherwood, B. Bhattacharjee, and A. Srinivasan. P5: A protocol for scalable anonymous communication. In *Proc. of the IEEE Symposium on Security and Privacy, 2002*, Oakland, CA, USA.
[28] D. Stinson. Something about all or nothing transforms. *Designs, Codes and Cryptography*, pages 133–138, March 2001.
[29] L. Subramanian. *Decentralized Security Mechanisms for Routing Protocols*. PhD thesis, University of California, Berkeley, Nov. 2005.
[30] M. Wright, M. Adler, B. Levine, and C. Shields. An analysis of the degradation of anonymous protocols. In *Proc. of NDSS 2002*, San Diego, CA, USA.
[31] L. Zhuang, F. Zhou, B. Y. Zhao, and A. Rowstron. Cashmere: Resilient anonymous routing. In *Proc. of NSDI 2005*, Cambridge, MA, USA.

Notes

[1]Elements of $\vec{I_x}$ and A belong to a finite field F_{p^q} where p is a prime number and q is a positive integer. All operations are therefore defined in this field and differ from conventional arithmetic.

[2] The entropy of a random variable x is $H(x) = -\sum_x P(x)log(P(x))$, where $P(x)$ is the probability function.

SAAR: A Shared Control Plane for Overlay Multicast

Animesh Nandi$^{\ddagger\diamond}$ *Aditya Ganjam*† *Peter Druschel*$^{\diamond}$ *T. S. Eugene Ng*‡
Ion Stoica§ *Hui Zhang*† *Bobby Bhattacharjee*$^{\Phi}$

$^{\diamond}$*Max Planck Institute for Software Systems* ‡*Rice University*
†*Carnegie Mellon University* §*University of California, Berkeley* $^{\Phi}$*University of Maryland*

Abstract

Many cooperative overlay multicast systems of diverse designs have been implemented and deployed. In this paper, we explore a new architecture for overlay multicast: we factor out the control plane into a separate overlay that provides a single primitive: a configurable anycast for peer selection. This separation of control and data overlays has several advantages. Data overlays can be optimized for efficient content delivery, while the control overlay can be optimized for flexible and efficient peer selection. Several data channels can share a control plane for fast switching among content channels, which is particularly important for IPTV. And, the control overlay can be reused in multicast systems with different data plane organizations.

We designed and evaluated a decentralized control overlay for endsystem multicast. The overlay proactively aggregates system state and implements a powerful anycast primitive for peer selection. We demonstrate that SAAR's efficiency in locating peers reduces channel switching time, improves the quality of content delivery, and reduces overhead, even under dynamic conditions and at scale. An experimental evaluation demonstrates that the system can efficiently support single-tree, multi-tree and block-based multicast systems.

1 Introduction

Cooperative endsystem multicast (CEM) has become an important paradigm for content distribution in the Internet [11, 36, 8, 2, 30, 47, 27, 7, 50]. In CEM, participating endsystems form an overlay network and cooperatively disseminate content. As a result, the resource cost of disseminating content is shared among the participants. Unlike server-based approaches, the content source need not provide bandwidth and server resources proportional to the number of receivers; unlike IP multicast [14], no network layer support is required; and unlike commercial content-distribution networks [1], no contract with a provider is needed.

Numerous CEM systems are being proposed and deployed by research, industry and open source communities. The systems cover a range of designs, using single tree-, multi-tree- or mesh-based data dissemination approaches and various overlay maintenance algorithms.

Common to all CEM systems is the problem of selecting overlay neighbors for data dissemination. We will show that the policies and mechanism used to make this selection significantly affect the performance of CEM systems. An ideal peer selection mechanism can support sophisticated selection policies, enabling high-quality data paths and good load balance, while accommodating participants with heterogeneous capabilities. An efficient peer selection mechanism can scale to large groups and large numbers of groups. Lastly, a responsive mechanism allows the system to more rapidly respond to failures and node departures, and it allows nodes to quickly join and switch content channels. Fast channel switching, in particular, is critical to emerging IPTV applications [23].

We show that peer selection for CEM systems can be performed using an *anycast* primitive, which takes as arguments a constraint and an objective function. Among the participating nodes that satisfy the constraint, the primitive selects one that maximizes the objective function. Such an anycast primitive offers a single, unified mechanism for implementing diverse data plane policies.

Consider the following example of a simple data dissemination tree. Each node *n* needs to select a parent. The constraint would require that a prospective parent is not a descendant of *n* and has spare forwarding capacity. Among the eligible nodes, the objective function might minimize the loss rate or the distance of the parent from the root. Much more complex data plane structures can be expressed in this way, e.g., multiple interior-node-disjoint trees as in SplitStream [7].

We have designed and evaluated *SAAR*, a *control overlay* for CEM systems. SAAR provides a powerful anycast primitive for selecting peers in one or more separate data dissemination overlays. SAAR provides several key benefits:

• *SAAR separates control and data dissemination into different overlay networks.* As such, it avoids a tradeoff between data and control efficiency. We show that the benefits of this separation outweigh the costs of maintaining a separate control overlay. First, the SAAR control overlay is optimized for efficient peer selection. When compared to current CEM systems, SAAR can locate more appropriate peers, and can do so faster. Rapid peer selection results in faster channel join and switching times; more appropriate peer selection improves data paths and delivery quality. Second, the data overlay is not constrained by a control overlay structure, and can therefore be optimized solely for efficient data dissemination and load balance, subject to application policy.

• *SAAR can support different data plane structures* (e.g. tree, multi-tree, mesh-based). Specific structures can be achieved by defining appropriate constraints and objective functions for anycast. Thus, SAAR separates the common control mechanism from the specific polices for maintaining a data overlay. As a reusable control overlay, SAAR simplifies the design of CEM systems.

• *A single SAAR overlay can be shared among many data overlay instances.* SAAR allows nodes to remain in the control overlay independent of their data channel membership. Control overlay sharing allows nodes to quickly join a channel and to rapidly switch between channels, which is critical for applications like IPTV [23].

The implementation of SAAR is layered on a structured overlay network [39, 8]. For each data overlay instance, a tree is embedded in this structured control overlay that connects the members of that data overlay. State information for members of a data channel is aggregated and disseminated within the corresponding tree. An anycast traverses the tree to locate peers for the data overlay, subject to the constraint and objective function. The aggregated state information is used to guide the search.

The rest of this paper is organized as follows. Section 2 briefly reviews existing CEM systems and other related work. Section 3 presents our proposed architecture and the design of SAAR. Section 4 describes how different data plane organizations can be built using SAAR. Section 5 presents an experimental evaluation of our SAAR prototype. We conclude in Section 6.

2 Background and related work

In this section, we consider existing CEM systems and other related work. The data planes of CEM systems can be classified as either path-based (single tree and multiple tree) or block-based. Path-based systems maintain one or more loop free paths from the content source to each member of a group. ESM [10], Overcast [25] and NICE [2] form a single tree, while SplitStream [7] and Chunkyspread [44] form multiple trees. In Bullet [27], CoolStreaming [50] and Chainsaw [31], the streaming content is divided into fixed-size blocks and group members form a mesh structure. Mesh neighbors exchange block availability information and swap missing blocks. Next, we discuss existing CEM systems from the perspective of the type of overlay network they utilize.

Unstructured overlay CEM systems construct an overlay network that is optimized primarily for data dissemination. Overlay neighbors are chosen to maximize the quality of the content delivery (i.e., minimize packet loss, delay and jitter), to balance the forwarding load among the overlay members, and to accommodate members with different amounts of network resources. Typically, a separate overlay is constructed for each content instance, consisting of the set of nodes currently interested in that content. The control plane is then implemented within the resulting overlay. Although these systems enable efficient data dissemination, the overlay is not optimized for efficient control.

Overcast [25], Host Multicast [49] and End System Multicast (ESM) [10] form a single dissemination tree. In the former two systems, nodes locate a good parent by traversing the tree, starting from the root. These protocols do not scale to large groups, since each member must independently explore the tree to discover a parent, and the root is involved in all membership changes. ESM uses a gossip protocol to distribute membership information among the group members. Each node learns a random sample of the membership and performs further probing to identify a good parent. The protocol is robust to node departures/failure but does not scale to large group sizes, where the membership information available to a given node tends to be increasingly partial and stale.

Chunkyspread [44] uses a multi-tree data plane embedded in an unstructured overlay, using a randomized protocol to select neighbors. The selection considers the heterogeneous bandwidth resources of nodes and assigns them an appropriate node degree in the overlay.

Bullet [27], CoolStreaming [50] and Chainsaw [31] use block-based data dissemination in an unstructured mesh overlay. Bullet has separate control and data planes. The control plane is not shared among multiple data channels and it was not designed to support different data plane organizations.

Structured overlay CEM systems use a structured overlay network [39, 35, 42, 38]. The key-based routing primitive [13] provided by these overlays enables scalable and efficient neighbor discovery.

In general, data is disseminated over existing overlay links. This constraint tends to make it more difficult to optimize data dissemination and to accommodate nodes with different bandwidth resources [3]. Group membership changes, on the other hand, are very efficient and the systems are scalable to very large groups and large numbers of groups in the same overlay.

Scribe [8], Subscriber/Volunteer(SV) trees [15] and SplitStream [7] are examples of CEM systems based on structured overlays. Scribe embeds group spanning trees in the overlay. The trees are then used to anycast or multicast within the group. Due to the overlay structure, some nodes may be required to forward content that is not of interest to them. SV trees are similar to Scribe, but ensure that only interested nodes forward content.

SplitStream uses multiple interior-node-disjoint dissemination trees that each carry a slice of the content. Compared to single-tree systems, it better balances the forwarding load among nodes and reduces the impact of node failures. SplitStream has an anycast primitive to locate parents with spare capacity in the desired trees when none can be found using Scribe. In SplitStream, however, this primitive is used only as a last resort, since it may add non-overlay edges and may sacrifice the interior-node-disjointedness of the trees.

NICE [2] is not based on a structured overlay as we defined it. Nevertheless, it shares the properties of efficient control but constrained data dissemination paths. Nodes dynamically organize into a hierarchy, which is then used to distribute the data.

Other related work: Anycast was first proposed in RFC 1546 [32]. GIA [26] is an architecture for scalable, global IP anycast. Both approaches share the drawbacks of network-layer group communication. That is, they require buy-in from a large fraction of Internet service providers to be effective at global scale, and they cannot easily consider application-specific metrics in the server selection. Application-layer anycasting [4] defines anycast as an overlay service.

Anycast within a structured overlay network has been used in several systems for decentralized server selection [9, 24, 41, 17]. Scribe [9] and DOLR [24] deliver anycast requests to nearby group members. Unlike SAAR, they provide only a coarse-grained overload protection mechanism by requiring overloaded group members to leave the group temporarily. *i*3 [41] provides fine-grained load balancing of anycast requests among the group members, but is not designed for efficient server selection based on multiple metrics like load, location and server state. Server selection in Oasis [17] is primarily optimized for locality, but also incorporates liveness and load. Oasis does not optimize the anycast based on proactive aggregation of state information. Unlike these systems, SAAR provides general and efficient anycast for peer selection in CEM systems.

Several systems use structured overlays for efficient request redirection [16, 46, 12]. CoDeeN [46], a cooperative CDN, distributes client requests to an appropriate server based on factors like server load, network proximity and cache locality. Coral [16] is a peer-to-peer web-content distribution network that indexes cached web pages and redirects client requests to nearby peers that have the desired content cached.

SDIMS [48] (influenced by Astrolabe [37]) aggregates information in large scale networked systems and supports queries over the aggregated state of a set of nodes. Internally, SDIMS relies on aggregation trees embedded in a structured overlay to achieve scalability with respect to both the number of nodes and attributes. SAAR implements a subset of of SDIMS's functionality, which is specialized for the needs of a CEM control plane.

ChunkCast [12] provides a shared control overlay, in which it embeds index trees for objects stored in the overlay. An anycast primitive discovers a nearby node that holds a desired object. ChunkCast is intended for block dissemination in a swarming file distribution system, and not for streaming multicast. Its anycast primitive is specialized for this purpose, and not for peer selection in a CEM system.

Pietzuch et al. [33] observe that structured overlays do not produce a good candidate node set for service placement in Stream-based overlay networks (SBONs). This is closely related to our observation that structured overlay CEM systems have constrained and sub-optimal data distribution paths.

Opus [5] provides a common platform for hosting multiple overlay-based distributed applications. Its goal is to mediate access to wide-area resources among multiple competing applications, in a manner that satisfies each application's performance and reliability demands. SAAR, on the other hand, provides a control overlay and an anycast peer selection service for a specific application, CEM. Thus, Opus and SAAR address largely complementary problems.

3 Design of SAAR

We begin with an overview of the SAAR control plane and describe its design in detail. Figure 1 depicts the SAAR architecture.

Our architecture for CEM systems separates the control and data planes into distinct overlay networks. There are no constraints on the structure of the data plane: it can be optimized for efficient data dissemination, can accommodate heterogeneity and includes only nodes that are interested in the content. The control overlay can be shared among many data plane instances, each disseminating a different content type or channel.

SAAR uses a decentralized control plane based on a structured overlay network. Its anycast primitive supports efficient and flexible selection of data dissemination peers. The SAAR overlay performs efficient, proactive state dissemination and aggregation. This aggregate state is used to increase the efficiency of the anycast primitive.

All nodes that run a particular CEM system participate in the SAAR control overlay, regardless of which

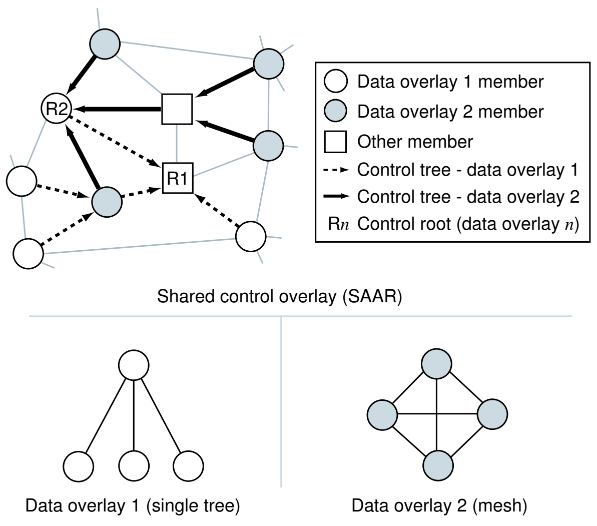

R2
R1
○ Data overlay 1 member
● Data overlay 2 member
□ Other member
Control tree - data overlay 1
Control tree - data overlay 2
Rn Control root (data overlay n)
Shared control overlay (SAAR)
Data overlay 1 (single tree)
Data overlay 2 (mesh)

Figure 1: SAAR architecture: Each node is a member of the control overlay and may be part of one or more data overlays. The members of a given data overlay are part of a tree embedded in the control overlay. Nodes use the SAAR anycast primitive to locate data overlay neighbors.

content they are currently receiving. This enables rapid switching between content channels. Even nodes that do not currently receive any content may choose to remain in the control overlay. In this "standby" mode, a node has low overhead and can join a data overlay with very low delay. As a result, membership in the control overlay is expected to be more stable and longer-term than the membership in any data overlay. Additionally, the sharing of state information across data overlays can reduce overhead, e.g., when a node is in more than one data overlay because it receives several content channels.

Group abstraction: The key abstraction provided by SAAR is a *group*. A group represents a set of nodes that are members of one data overlay. The group's control state is managed via a spanning tree that is embedded in the control overlay and rooted at a random member of the control overlay. Due to the SAAR overlay structure, the spanning tree may contain interior nodes that are not part of the group. The group members may choose to form any data overlay structure for data dissemination.

A set of *state variables* is associated with a group. Each group member holds an instance of each state variable. Typical examples of state variables are a node's forwarding capacity, current load, streaming loss rate, tree-depth in a single-tree data plane, etc.

SAAR can aggregate state variables in the spanning tree. Each state variable g is associated with an *update propagation frequency* f_{up}, a *downward propagation frequency* f_{down} and an *aggregation operator* A. The values of a state variable are periodically propagated upwards towards the root of the group spanning tree, with frequency at most f_{up}. (The propagation is suppressed if the value of a variable has not changed). At each interior node, the values received from each child are aggregated using the operator A. The aggregated value at the root of the spanning tree is propagated down the tree with frequency at most f_{down}. State variables for which no aggregation operator is defined are propagated only one level up from the leaf nodes.

The aggregated value of g (using aggregation operator A) at an intermediate node in the spanning tree is denoted by g^A. For example, the value of g_{cap}^{SUM} at the root of the spanning tree would denote the total forwarding capacity of the group members.

Anycast primitive: SAAR provides an *anycast* operation that takes as arguments a *group identifier* G, a *constraint* p, an *objective function* m, and a *traversal threshold* t.

The primitive "inspects" group members whose state variables satisfy p and returns the member whose state maximizes the objective function m among the considered members. To bound the anycast overhead, at most t nodes are visited during the tree traversal. Note that a search typically considers many more nodes than it visits, due to the propagation of state variables in the tree. If $t = \perp$, the first considered node that satisfies the predicate is selected.

The predicate p over the group's state variables specifies a constraint on the neighbor selection. Typically, the constraint is chosen to achieve the desired structure of the data overlay. A simple example predicate $p = (g_{load} < g_{cap})$ would be used to locate a node with spare forwarding capacity.

The anycast selects, among the set of nodes it inspects and that satisfy p, a node whose state variables maximize the objective function m. m is an expression over the group's state variables and evaluates to a numeric value. For example, using the state variable g_{depth} to denote the tree depth of a member in a single-tree data plane, the objective function $m = 1/g_{depth}$ would select a node with minimum depth in the tree, among the considered nodes that satisfy the predicate $p = (g_{load} < g_{cap})$.

The anycast primitive performs a depth-first search of the group spanning tree, starting at the requester node. It uses a number of optimizations. If the aggregated state variables of a group subtree indicate that no member exists in the subtree that is both eligible (i.e., satisfies the constraint) and superior (i.e., achieves a better value of the objective function than the current best member), then the entire subtree is pruned from the search. Similarly, if the aggregated state variables of the entire tree (propagated downward from the root) indicate that no eligible and superior member exists in the tree, then the anycast terminates immediately.

create(G, set of (g_v,A_v,f^v_{up},f^v_{down})).	Creates a group with its group variables, their aggregation operators and propagation frequencies.
join(G)	This function is called by a node that wishes to join the group G.
anycast(G, p, m, t)	This function is called by a node to select a member of the group G. p is the constraint, m is the objective function, t is the maximal number of nodes visited.
update(G, set of g_v)	Called by a node to update the group with the current values of its state variables.
groupAggregateRequest(G, g_v)	Returns the value of the aggregated state variable g_v at the root of the spanning tree.
leave(G)	Called by a node that wishes to leave the group G.

Table 1: SAAR API

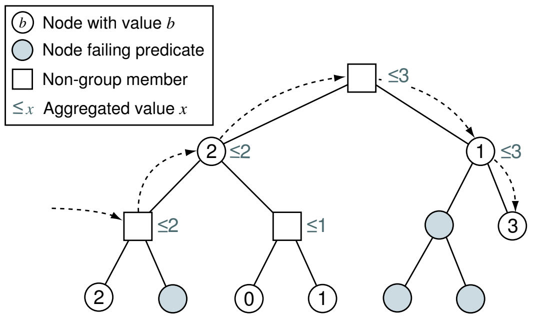


Figure 2: Anycast traversal example: Given an anycast request issued at the leftmost interior node in the group spanning tree, the anycast traverses the tree in a depth-first search. The search only visits subtrees with members that satisfy the predicate and whose value exceeds that of the current best known member.

Since the SAAR overlay construction is proximity-based, the group members are inspected roughly in order of increasing delay from the requester node *n*. Therefore, the result is chosen from the nodes with least delay to *n*, among the nodes that satisfy the constraint. This bias can be removed by starting the anycast traversal instead from a random member of the SAAR control overlay[1].

There is an inherent trade-off between accuracy and scalability in the distributed selection of overlay neighbors. Accuracy is maximized when decisions are based on complete and current information. To maximize accuracy, either (1) all nodes maintain current information about many nodes in the system, or (2) an anycast visits many nodes in the system for each peer selection. Neither approach is scalable. SAAR mitigates this tension by propagating aggregated state and by limiting the scope of anycast tree traversals based on this state. Also, in SAAR, accuracy and overhead can be controlled by bounding the anycast overhead (threshold t), and by changing the propagation frequencies of the state variables.

Example anycast traversal: Figure 2 shows an example group spanning tree. A new node wants to join the data overlay and seeks a parent that maximizes the value of an integer state variable among the nodes that satisfy a given constraint. There are six members that satisfy the constraint. Given an anycast request issued at the leftmost interior node in the spanning tree, the anycast traverses the tree in a DFS, pruning subtrees that contain no eligible members with a value of the variable that exceeds that of the current best known member. In the example shown, the anycast stops after visiting five nodes, and yields the rightmost leaf node with the value 3. Had the anycast been invoked with a value of $t < 5$, then the anycast would have stopped after visiting t nodes, and yielded the leftmost leaf node with value 2.

[1]Here, we route the anycast message to the node responsible for a randomly drawn key in the underlying structured overlay network.

SAAR API: The SAAR API contains functions to create, join and depart groups, to locate a neighbor, retrieve pre-computed aggregated group state, and to update the control plane with a member's group variables. The operations are listed in Table 1.

3.1 Implementation

The implementation of SAAR uses Scribe [8, 9] to represent groups and to implement the anycast primitive. Scribe is a group communication system built upon a structured overlay network. Each group is represented by a tree consisting of the overlay routes from group members to the node responsible for the group's identifier.

Due to proximity neighbor selection (PNS) in the underlying Pastry overlay [6, 21], the spanning trees are proximity-based, i.e., nodes within a subtree tend to be close in the underlying Internet. An anycast primitive walks the tree in depth-first order, starting from a node that is close to the anycast requester until an appropriate node is found [9].

Scribe does not have group variables and does not aggregate or propagate state in a group tree. Scribe's anycast primitive does not take a constraint or objective function. Our implementation adds support for these facilities.

Nodes in a SAAR group tree aggregate and propagate state variables up and down the tree, similar to SDIMS [48]. To reduce message costs, update messages are propagated periodically, they are combined on a given overlay link across state variables and across group trees, they are piggy-backed onto other control traffic when possible, and they are multicast down the tree. During an anycast, a DFS traversal of the group tree prunes entire subtrees based on the aggregated state of the subtree. In addition, we implemented the following optimizations:

Using network coordinates: SAAR employs virtual network coordinates to guide the depth-first search in the group trees. We used NPS [29] with a 5-D coordinate system in our implementation. Specifically, network coordinates are used to visit an interior node's children in order of increasing distance to the anycast requester. Moreover, node coordinates can be exported as state variables; thus, they can be used to guide the selection of nodes based on their location.

Multiple spanning trees: To increase the robustness to churn in the control overlay, SAAR maintains multiple interior-node-disjoint spanning trees connecting the members of each group. Thus, the failure or departure of a node may cause a subtree disconnection and subsequent repair in at most one tree. By starting multiple traversals in different trees in parallel, anycast operations can be served in a timely fashion even while one of the spanning trees is being repaired. Interior-node-disjoint trees are constructed in Scribe simply by choosing group ids that do not share a prefix, as in SplitStream [7].

Our SAAR prototype was implemented based on the FreePastry implementation of Scribe [18]. Implementing SAAR added 4200 lines of code. Implementing a single-tree, multi-tree, and block-based CEM based on SAAR, as described in the following section, added another 1864, 2756 and 3299 lines of code, respectively.

4 Using the SAAR Control Overlay

This section describes how the SAAR control overlay can be used to construct CEMs with single-tree, multiple-tree, and block-based data overlays.

4.1 Single-Tree Multicast

To implement a single-tree multicast protocol using SAAR, each data overlay instance is associated with a SAAR group. Data overlay neighbors are selected such that (i) the neighbor has spare forwarding capacity and (ii) adding the neighbor does not create a loop in the data path. In addition, the control plane should preferentially select neighbors that (i) experience low loss, (ii) are near the requester and (iii) have low depth in the tree. These requirements can be expressed via the constraint and the objective function arguments to a SAAR anycast. Assuming the data stream has rate BW_{stream} and a node's forwarding bandwidth is BW_{node}, we define the forwarding capacity of a node as $D = BW_{node}/BW_{stream}$.

The group associated with our single-tree data plane uses the following state variables, constraint and objective function:

• *State Variables:* $g_{cap} = D$ is the maximum number of children a node can support; g_{load} is the current number of children, g_{path} is a list of node identifiers on the node's path to the root, g_{depth} is the length of the node's path to the root, g_{loss} is the streaming loss rate, and g_{pd} is the path delay from the root. No aggregation operator is defined for g_{path}.

• *Constraint:* A requesting node r selects a prospective parent that has free capacity, will not cause a loop and has a loss rate less than a threshold L, using the predicate:

$$(g_{load} < g_{cap}) \wedge (r \notin g_{path}) \wedge (g_{loss} < L)$$

Alternatively, the term $(g_{loss} < r_{loss})$ can be used to select a parent that has lower loss than the requester.

• *Objective Function:* The objective function is either $1/g_{depth}^{MIN}$ or $1/g_{pd}^{MIN}$, which minimizes depth and path delay, respectively, as motivated by the findings in [40].

The source of a multicast event creates a new group and then joins it. An interested node calls the anycast method to locate a parent. Once it receives data, it joins the group, allowing the control plane to select it as a potential parent. The node uses the update method to inform the control plane of the current values of its state variables. To leave, a node disconnects from its parent and leaves the group. When a node fails or leaves, its children select a new parent using the anycast method.

Periodic data plane optimization: When a node joins or recovers from a disconnect, it uses a traversal threshold $t = \perp$ to find an eligible parent as quickly as possible.

The system gradually improves the tree's quality by periodically (e.g. every 30 seconds) anycasting with a traversal threshold of $t = 2\log_k N$, where N is the approximate size of the group and k is the arity of the control tree. It can be shown that this anycast considers at least $n = k^{\frac{-1+\sqrt{1+8t}}{2}}$ nodes. Assuming that the eligible nodes are uniformly distributed in the control tree, the probability is greater than $(1-(f/100)^n)$ that we find a peer in the fth percentile of eligible nodes, sorted by decreasing value of the objective function. Thus, a value of $t = 2\log_k N$ ensures that a "good" node is found with high probability. With a system size of at least 1024 and a control tree arity of $k = 16$, for instance, we locate a peer in the 90% percentile of eligible nodes with three-nines probability.

Preemption: If a node r with a forwarding capacity $r_{cap} > 0$ is disconnected and cannot locate a new parent, r uses an anycast to locate a parent that has a child with no forwarding capacity. (Such a child must exist, else there would be leaf nodes with spare capacity). This is done using boolean group variable g_{zdc}, which is true when a node has a zero-degree child. The anycast takes the modified predicate:

$$(g_{zdc} \vee (g_{load} < g_{cap})) \wedge (r \notin g_{path}) \wedge (g_{loss} < L)$$

Once such a parent is located, the node preempts the zero-degree child, attaches to the selected parent and adopts the preempted node as its child.

4.2 Multi-Tree Multicast

Next, we built a multi-tree CEM system similar to SplitStream [7] using SAAR. SplitStream was designed to more evenly balance the forwarding load and to reduce the impact of node and network failures, relative to a single-tree CEM. The content is striped and disseminated using k separate, interior-node-disjoint distribution trees, where each stripe has $1/k$-th of the stream bandwidth. The constraint, objective function and state variables are the same as in the single-tree CEM. However, there is an instance of each variable per stripe. We use a single SAAR group per multi-tree data overlay, with the following state variables: $g_{coord}[i]$, $g_{cap}[i]$, $g_{load}[i]$, $g_{path}[i]$, $g_{loss}[i]$, $g_{depth}[i]$, $g_{pd}[i]$, $i \in [0, k-1]$.

A node forwards data (i.e., accepts children) only in its *primary* stripe ps. This construction ensures interior-node-disjoint stripe trees: a node is an interior node in at most one stripe tree and a leaf in all other stripe trees. Thus, a node with forwarding capacity D (defined in Section 4.1) has

$g_{cap}[ps] = D * k$ and $g_{cap}[i] = 0, \forall i \neq ps$.

In SplitStream, the primary stripe selection is fixed by a node's identifier to allow the efficient construction of interior-node-disjoint stripe trees. This can lead to a resource imbalance when the node forwarding capacities are heterogeneous. In the SAAR-based implementation, nodes can choose their primary stripe so as to balance the available forwarding capacity in each stripe. To do this, a joining node selects as its primary the stripe with the least total forwarding capacity at that time[2]. A node uses the aggregated value of the state variables g_{load} and g_{cap} and chooses ps to be the i that minimizes

$(g_{cap}[i]^{SUM} - g_{load}[i]^{SUM})$.

Even with this adaptive choice of a primary stripe, it is still possible that the departure of a node causes a stripe to be momentarily left with no forwarding capacity, until another node joins. As in SplitStream, a number of tree transformations are possible in this case [7], which can be easily expressed in SAAR. As a last resort, a child relaxes the predicate to select a parent with forwarding capacity in a different stripe, at the expense of interior-node-disjointedness. The system behaves like SplitStream in this respect, except that flexible primary stripe selection significantly reduces the likelihood of stripe resource exhaustion.

Moreover, the SAAR-based implementation can support any number k of stripes, allowing the choice to match the needs of the application coding scheme. To achieve good load balance in SplitStream, on the other hand, the number of stripes must correspond to the routing base in SplitStream's underlying Pastry overlay, which is a power of 2. The flexible choice of k, and the flexible primary stripe selection, are two examples where the power of SAAR's anycast primitive makes it possible to relax constraints on the data plane construction in the original SplitStream implementation.

The constraint and objective function used to locate parents now apply on a per stripe basis. For example, to locate a parent in stripe s, the corresponding predicate is

$(g_{load}[s] < g_{cap}[s]) \wedge (r \notin g_{path}[s]) \wedge (g_{loss}[s] < L)$.

[2]Note that with a coding scheme like MDC [20], the stripes are equivalent from the perspective of the application.

4.3 Block-Based Multicast

In block-based CEMs [50, 31], a random mesh connects the members of the data overlay, and a swarming technique is employed to exchange blocks amongst the mesh neighbors.

We use SAAR to select and maintain mesh neighbors, rather than the commonly used random walk [45] or gossiping [19] techniques. In Section 5.4, we will briefly describe the swarming algorithm (based on existing literature) we have implemented in our block-based prototype.

Mesh Neighbor Selection: A member n with forwarding capacity D (defined in Section 4.1) maintains between M (e.g., 4 as in Coolstreaming [50]) and $M * D$ neighbors. In steady state, a node expects to receive $1/M$ of the total stream bandwidth from a particular neighbor; thus the minimum number of neighbors is M. Nodes use SAAR anycast to maintain M neighbors of good quality and accept up to $D * M$ neighbors.

To ensure that the mesh construction has sufficient path diversity, we anycast with $t = \perp$ but start from a random group member, so that the selected node is not necessarily near the requester. In addition, each node periodically locates fresh neighbors, even if it has M neighbors of good quality. We have observed that without this periodic update, nodes that joined early tend to have their neighbor capacity exhausted and thus they lack links to nodes that joined much later, resulting in a low path diversity and high depth.

A SAAR group associated with a block-based data plane uses the following state variables and constraint (no objective function is used):

- *State Variables:* g_{cap} is the maximum number of neighbors ($D * M$), g_{load} is the current number of neighbors, g_{loss} is the loss rate.
- *Constraint:* The predicate is $(g_{load} < g_{cap}) \wedge (g_{loss} < L)$.

Note that the loop-freedom constraint needed in tree-based systems is not present.

Channel Join Delay	The delay from the instant a node joins a multicast event until it receives 90% of the stream rate.
Tree Depth	The depth of a node in the dissemination tree.
Continuity Index	The fraction of the unique data packets streamed during a node's membership that were received by the node.
Datastream Gap	The time during which a node fails to receive data due to the departure of a node in the data plane.
Node Stress	Total number of control messages (not data messages) sent and received per second, per node.

Table 2: Evaluation metrics.

5 Experimental Evaluation

We begin with a description of the experimental setup. With the exception of the Planetlab [34] experiments in Section 5.5, we use Modelnet [43] to emulate wide-area delay and bandwidth in a cluster of PCs connected by Gigabit Ethernet, each with a 2.6 Ghz CPU and 4 GB of main memory. We chose Modelnet because it allows a meaningful comparison of various systems and protocols, as we can deterministically reproduce the network conditions for each experiment.

Using up to 25 physical cluster nodes, we emulate a network with 250 stubs. (The Modelnet core ran on a separate cluster node.) The delays among the stubs were randomly chosen from the King [22] data set of measured Internet delay data. Four client nodes are attached to each stub network for a total of 1000 client nodes. The client nodes are connected via 1 ms links to their respective stubs. Neither the access links nor the stub network were the bottleneck in any of the experiments. Similarly, we ensured that the CPU was not saturated during the experiments on any of the cluster nodes.

We emulated overlays of 250–900 virtual nodes. To emulate an overlay of size n, we randomly selected n client nodes to participate in the overlay. The forwarding capacity (as defined in Section 4.1) of virtual nodes was limited via their node degrees. The node degrees are heterogeneous and follow the measured distribution from the Sripanidkulchai et al. study of live streaming workloads [40].

However, we use a minimum node degree of one in the experiments to ensure that some forwarding capacity is always available during random node joins and departures. Also, we impose a maximum degree cap to achieve a given mean Resource Index (RI), where mean RI is the ratio of the total supply of bandwidth to the total demand for bandwidth. Unless stated otherwise, we use degree caps of (MIN=1,MAX=6) to achieve a mean RI=1.75. The degree distribution after enforcing the caps is as follows: approximately 76.85% of degree 1, 9.5 % of degree 2, 0.34% each of degree 3, 4 and 5, and 12.4% of degree 6. Unless stated otherwise, the multicast source had a degree of 5.

In the Modelnet experiments, we streamed data from a single source node at a constant rate of 32 Kbps. We chose this low rate to reduce the load on the Modelnet emulation. This does not affect the results, since we are interested in control efficiency and its impact on the quality of the data streams. Since the streaming rate is identical in all systems and we are primarily interested in control overhead, we exclude data packets when measuring message overheads. We evaluate the performance of the various systems using the metrics described in Table 2.

In all experiments, a single SAAR control overlay is used that includes all participating nodes, irrespective of their data overlay membership. We use a single spanning tree per SAAR group in scenarios without control overlay churn, and two trees per group otherwise. All reported results are the averages of at least 2 runs. Error bars, where shown, indicate the minimal and maximal measured values for each data point. In cases where no error bars are shown in the plots, the deviation among the runs was within 3%.

5.1 Effectiveness of SAAR Anycast

Our first set of experiments evaluate the performance of SAAR's anycast primitive. No data was streamed in these experiments.

Locality-awareness: We evaluate the anycast primitive's ability to find peers with low delay. We run SAAR with a traversal threshold t of $\perp$ and 2, respectively. To isolate the effects of using NPS coordinates during tree traversal, we evaluate SAAR with and without NPS coordinates (SAAR-NO-NPS), and compare its performance against a centralized system where peers are chosen either randomly (CENTRAL-Random) or using NPS coordinates (CENTRAL-NPS). CENTRAL-Global reflects the optimal greedy peer selection based on global knowledge.

We use a 250 node overlay and 10 groups. Peers subscribe to each group with a probability of 0.1, resulting in an expected group size of 25 peers. Figure 3 shows that SAAR's ability to select nearby peers comes close to that of a centralized solution that uses NPS coordinates. Using NPS in the tree traversal significantly improves the results, though even the result without NPS (corresponding to a plain Scribe anycast) is significantly better than random peer selection.

Load awareness: Next, we evaluate SAAR's ability to quickly select peers with available bandwidth under conditions with a low Resource Index (RI), where there are few nodes with spare bandwidth. Figure 4 compares SAAR with a centralized peer selection service, while building a single-tree data overlay of $N = 350$ nodes. The centralized peer selection service was placed on a node

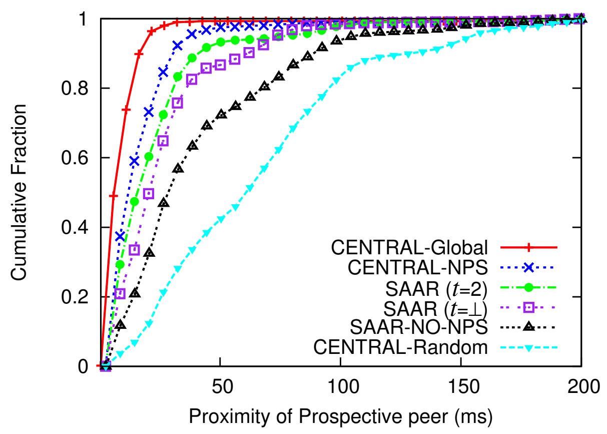


Figure 3: Locality awareness

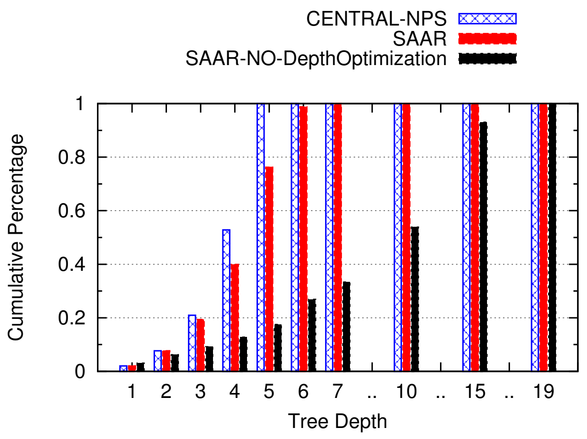


Figure 5: Tree depth optimization

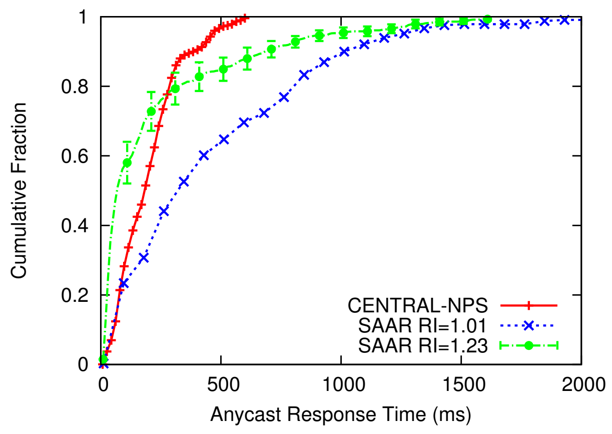


Figure 4: Load awareness

such that the average delay to the remaining nodes in the underlying 250-node stub network was minimized.

We experimented with RI=1.01, in which all nodes have exactly degree 1, (except the source, which has degree 5) and another setting of RI=1.23 with degree cap (MIN=1,MAX=2). Even under the harsh RI=1.01 setting, SAAR anycast can select a peer within 1 second in 90% of the cases. This is because SAAR's anycast tree traversal prunes subtrees with no capacity based on aggregated information. When a moderate amount of spare bandwidth is available (RI=1.23), 78% of the anycast response times are even lower than those of the centralized server, because SAAR's anycast can usually find a peer by contacting a node that is closer than that server.

During the experiment, the average/median/99th percentile number of tree nodes visited during an anycast was 3.2, 3, and 4 with RI=1.01, and 2.3, 2, and 4 with RI=1.23. The 95th percentile and the maximum of the total message overhead during the experiment was less than 4 msgs/sec and 18 msgs/sec, respectively with RI=1.01; it was less than 3 msg/sec and 12 msgs/sec with RI=1.23.

The maxima were reached at the root node of the group tree in each case.

Tree depth optimization: The next experiment shows that SAAR anycast can optimize for metrics like tree depth effectively. We compare the achieved tree depth with that of a centralized membership server. 250 nodes join a channel over a period of 120 seconds. We set the maximum tree traversal threshold $t = 4$. During the traversal, we use aggregated state variables to prune subtrees in which the depth of the minimum depth peer with available capacity is greater than the best choice we have so far. Figure 5 shows that the tree depths resulting from using SAAR anycast are almost as good as those obtained with the centralized server. For comparison, we also included the result for the case when SAAR is being used without an objective function to minimize tree depth. Moreover, the anycasts had low traversal overhead. The 95th percentile and the maximum of the total message overhead during this experiment was less than 3 msgs/sec and 34 msgs/sec, respectively.

In summary, the decentralized SAAR control plane can effectively and efficiently select nearby nodes, nodes with spare capacity, and select nodes subject to metrics like tree depth.

Control overheads: Next, we present results of a simple analysis of SAAR's control overhead. Assume that we construct a SAAR overlay with N nodes using base-k routing in the underlying Pastry network. State variables are propagated once every second. There are G groups in the system, and the average group size is g. Define $T = g*G*\log_k N$; T is an upper bound on the number of edges in *all* the control trees in the system.

The aggregation analysis considers two cases (when $T \leq kN$ and when $T > kN$). If $T \leq kN$, then an upper bound on the average number of control messages sent and received per node per second due to state aggregation $S = 2*\frac{T}{N}$. If $T > kN$, then $S = 2*(k-1)*\log_k \frac{T}{N}$. Now let, on average, there be a anycasts per second per group in the system. The upper bound on the average number of anycast messages per node per second is simply $2 * \frac{\log_k N}{N} * a * G$.

Consider a large system with 10^6 nodes, a small number of large groups (ten groups of 10^5) and many small groups (10^5 groups of ten) in a SAAR overlay with $k = 16$. For an average node, the aggregation overhead in this case is no more than 20 msgs/sec. Even if we assume that each group turns over every 5 minutes, then the anycast overhead is less than $\frac{1}{7}$ msgs/sec for the average node.

In another configuration, assume that every node is a member of one group. Irrespective of the size distribution of the groups, $g * G = N$. Here, the average aggregation overhead is no more than 10 msg/sec and the corresponding anycast overhead is less than $\frac{1}{15}$ msg/sec in this case.

These results are consistent with our measured average node stress results. We note that the stress at nodes near the root of a control tree are significantly higher than average in our implementation. To some extent, this is inherent in any tree-based control structure. The difference between maximal and average node stress could be reduced by limiting the degree of nodes, at the expense of somewhat deeper trees. Also, the node stress tends to balance as the number of groups in the same SAAR overlay increases, because the group tree roots (and thus the nodes near the root) are chosen randomly.

5.2 SAAR for Single-Tree Data Overlays

Next, we show the benefits of using the SAAR control overlay in the design of a single-tree CEM. We compare the performance of the native single-tree ESM system [10] with a modified implementation of ESM using SAAR.

350 nodes join a single-tree CEM and continue to leave/rejoin the multicast channel with an exponentially distributed mean session time of 2 minutes and a minimum of 15 seconds. To achieve a large mean group size, the nodes rejoin the same multicast channel after an offline period of 10 sec. The experiment lasts for 1 hour. We compare the performance of four systems below. All systems attempt to minimize the tree depths while locating parents.

Native-ESM: We use ESM [10] as an example of a single-tree CEM based on an unstructured overlay. A single overlay is used for control and data. The overlay is optimized for data dissemination; state information is disseminated epidemically to enable peer selection.

Scribe: We use Scribe as an example of a single-tree CEM based on a structured overlay. A single overlay is used for control and data. Scribe's standard pushdown policy is used to enforce the degree bounds at the intermediate nodes in the Scribe tree [8].

SAAR-ESM: A version of ESM that uses a shared SAAR overlay for control. Nodes remain in the SAAR control overlay with an exponentially distributed session time with a mean of 30 minutes, for the experiment duration. As before, peers switch between data overlays with mean session time of 2 minutes and a minimum of 15 seconds. As our results show, nodes have an incentive to remain in the control plane longer than in any particular data channel, because it enables them to join and switch between channels much faster while the overhead is very low.

SAAR-ESM-Unshared: To isolate the benefits of a more stable control plane membership, we make nodes join/leave the SAAR control overlay whenever they join/leave a multicast group in this system. Otherwise, the system is identical to SAAR-ESM.

Figure 6 shows the results of our experiments. Among all systems, SAAR-ESM achieves easily the best results for join delay, continuity and node stress. SAAR-ESM-Unshared appears to beat SAAR-ESM in terms of tree depth. This comparison turns out to be misleading, however, because the average steady-state group size achieved by SAAR-ESM-Unshared during the experiment is only 55% that of SAAR-ESM's, due to the large difference in join delays in combination with churn. The average group sizes in the experiment are 225 (Native-ESM), 180 (Scribe), 290 (SAAR-ESM) and 160 (SAAR-ESM-Unshared), respectively.

The long tail in the SAAR-ESM join delay distribution corresponds to the initial joins when a node is not yet a member of the control overlay. Subsequent joins exhibit very low join delay: 99.8% of such joins had a delay of less than 1.5 sec. Native-ESM exhibits higher join delay and a lower continuity index than SAAR-ESM. Additional results (not shown) show that this gap widens with higher churn or larger groups. This is because in Native-ESM, state information propagates slowly, causing increasing staleness as churn or group size increases.

The results confirm earlier observations that Scribe exhibits deep trees and relatively high join delay under churn or when the node capacities are heterogeneous [3]. One reason is that the overlay structure imposes constraints on the data plane, resulting in Scribe pushdown being the norm rather than the exception. Another reason is that churn disrupts the coupled control and data overlay in Scribe. The combined effect is higher tree repair time, which leads to a poor continuity index.

The node stress incurred by SAAR-ESM is generally lowest among all systems, except for a longer tail. (The tail is a result of higher node stress near the top of the group tree.) This result indicates that the overall reduction in control churn in the shared SAAR overlay more

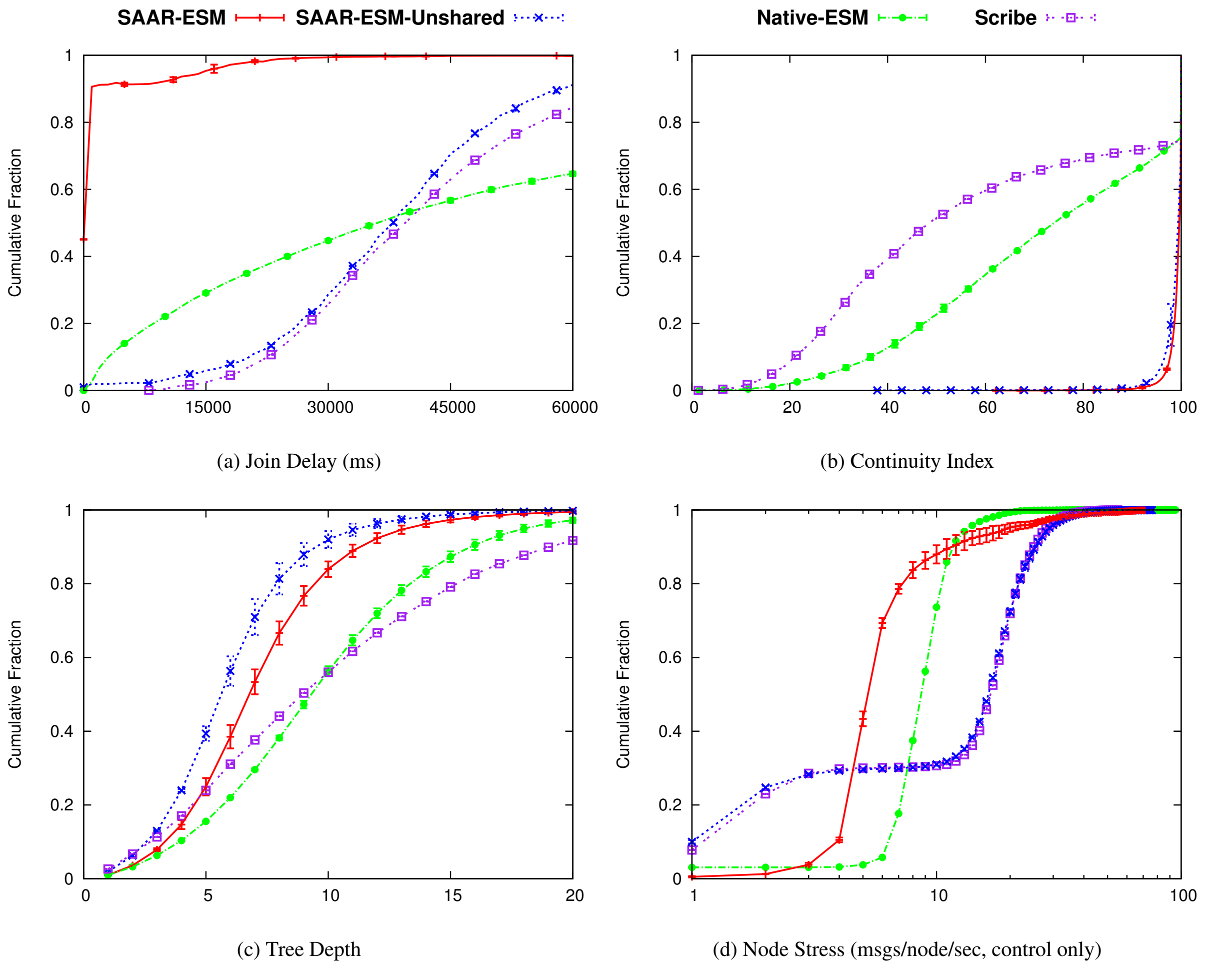


Figure 6: Single-tree CEM performance

than outweighs the additional overhead for maintaining a separate control overlay.

Comparing SAAR-ESM and SAAR-ESM-unshared confirms that a shared control overlay leads to lower join delays, better continuity and reduced node stress. Even SAAR-ESM-Unshared, however, yields dramatically better continuity than Native-ESM and Scribe. This speaks to the power of the SAAR anycast primitive, independent of the control plane sharing.

We believe that shielding the control plane from churn due to channel switching, as provided by a shared SAAR control overlay, is a critical optimization for applications like IPTV [23]. There, join and switching delays are very important and users switch among channels much more frequently than they start/stop their IPTV application.

We performed additional experiments comparing the performance of Native-ESM and SAAR-ESM with different levels of data and control overlay churn, and under flash crowd conditions. We also compared SAAR-ESM with a centralized membership service in terms of scalability. For lack of space, we only summarize the results here. More detailed results can be found in a technical report [28].

Lower Membership Churn: We repeated the previous experiment with a mean session time of 5 minutes. The join delays in Native-ESM improved significantly, with a 75th percentile of 15 sec. For SAAR-ESM, as before, 99.8% of joins where a node was already a part of the SAAR overlay completed within 1.5 secs. The continuity index of Native-ESM also improved significantly, with a 75th percentile of over 90. For SAAR-ESM, the 98th percentile continuity index improved to over 98.

These results show that Native-ESM, which was not designed for very high membership churn, performs well under less severe membership dynamics. However, even under these conditions, SAAR lends ESM significantly better performance.

Control overlay churn: We also evaluated SAAR-ESM with different levels of churn in the control overlay, and with one or two spanning trees per SAAR group.

Even at a mean control overlay session time of only 5 minutes (exponential distribution) and an overlay size of 350 nodes, using 2 trees per group yields anycast performance comparable to that of a single tree with no overlay churn. An exception are join events that involve a control overlay join: consistent with the results in Figure 6(a), these have a noticeably higher join delay. The overhead doubles when two trees are used, but the average overhead is still modest at 10 messages/second/node.

Flash crowds: A group of 350 nodes join an event within 15 seconds, and remain in the group for 15 minutes. For SAAR-ESM, the nodes are already part of the control overlay when the experiment starts.

SAAR-ESM is able to connect nodes to the data overlay quickly: the 90th percentile of the join delay distribution is less than 4 secs. The corresponding 90th percentile for Native-ESM is more than 8 secs. SAAR-ESM is able to maintain low tree depths, with an 80th percentile of 7 and a maximum of 8. For comparison, in Native-ESM, the 80th percentile and the maximum tree depths are both 8. In SAAR-ESM, 90% of nodes have a stress of less than 3, while Native-ESM has an average node stress of 21.

These results show that SAAR's anycast primitive yields significantly lower join delay and lower overhead than Native-ESM under flash crowd conditions, while maintaining comparable tree depth.

Scalability: We compare SAAR-ESM with a version that uses a centralized membership service. We use data overlays of sizes 54, 180, 540 and 900 nodes, all at a mean data session time of 2 minutes (exponential distribution) and a minimum of 15 secs. In SAAR-ESM, all nodes join the control overlay before the start of the experiment and remain in the overlay during the experiment.

The centralized membership service handled 50, 177, 662, 1084 messages/second at the various overlay sizes, showing an expected linear increase in the load. Due to the resulting bottleneck, the 90th percentile peer selection delays of the central membership service increases as 750 ms, 1.1 sec, 2.4 sec, 18 sec, respectively, for the different overlay sizes. For SAAR-ESM, the 90th percentile anycast delay increases from 600ms at 54 nodes to only 1.2 seconds at 900 nodes. The average continuity index achieved with the centralized membership service and SAAR-ESM at a group size of 900 was 80.3 and 97.6, respectively. This clearly demonstrates the scalability of the SAAR control plane.

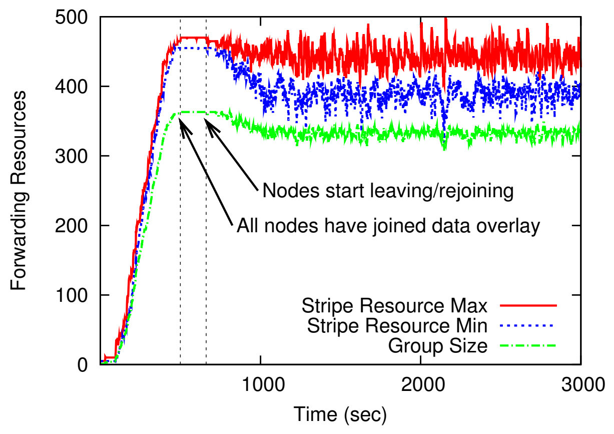
In summary, the results clearly show that SAAR's efficient anycast primitive yields ESM superior join delays, better content delivery quality, increased robustness to churn and increased scalability. Moreover, the shared SAAR control overlay dramatically reduces join delays and increases efficiency by reducing membership churn in the control plane. Additionally, these benefits are realized at a lower overhead than Native-ESM and Scribe. Comparing the SAAR-ESM with the native ESM system, we have shown that using a decoupled, shared control plane can achieve the best of both data dissemination quality and control efficiency. SAAR is effective in constructing high quality data overlays under flash crowd scenarios and high data overlay dynamics, while tolerating control plane churn. Finally, unlike a centralized membership service, the decentralized design of SAAR allows it to support data overlays of large size.

5.3 SAAR multi-tree CEM performance

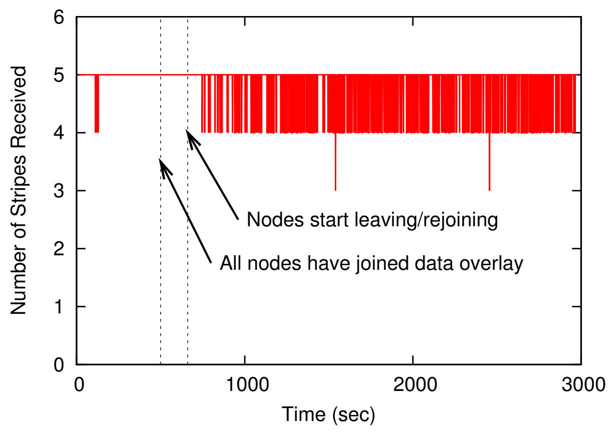
To show the effectiveness of SAAR in supporting multi-tree data planes, we have implemented a prototype multi-tree CEM based on SAAR, as described in Section 4.2. We use five data stripe trees per data overlay. 350 nodes with heterogeneous degree distribution (mean RI=1.23 and degree caps MIN=1,MAX=2) join the control overlay in the first 10 minutes, and then join the data overlay in the next 5 minutes. They remain in the data overlay for another 10 minutes, and then continue to leave/rejoin the data overlay with a mean session time of 2 minutes (exponential distribution) and a minimum of 15 seconds for the remainder of the experiment. To achieve a large instantaneous group size, nodes re-join the same data overlay 10 seconds after leaving. Nodes do not depart from the control overlay during the experiment, which lasted for approximately one hour. We show that SAAR can effectively balance resources among the stripe trees despite constrained resources and heterogeneous node degrees. We also measure the resulting join delay, continuity index and control overhead.

Figure 7(a) shows the instantaneous group size, as well as the minimum and the maximum of the total forwarding resources among the stripes. The minimum resources are always above the demand, i.e., the group size. The fluctuations in stripe resources result from membership churn. For instance, when a node of degree 6 leaves, the capacity in its primary stripe drops by $6*5=30$ units. In all cases, however, the imbalance is quickly rectified due to the adaptive primary stripe selection policy for newly joining nodes.

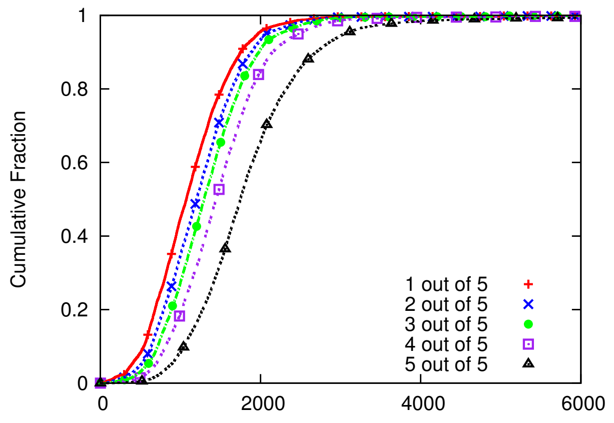
Figure 7(b) plots, for each data sequence number, the minimum number of stripes that 95% of the nodes are able to receive. Every second, the multicast source increments the sequence number. For each sequence number, there are 5 data packets generated, one for each stripe. Thus, a value of 4 stripes received means that 95% of the nodes are able to receive 4 or more stripes.



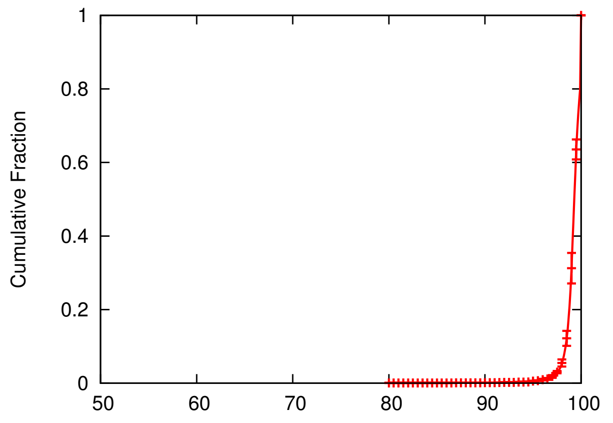
(a) Resource Balance



(b) Stripes Recieved



(c) Join Delay (ms)



(d) Continuity Index

Figure 7: SAAR multi-tree CEM performance

Figure 7(c) shows the CDF of the join delay of nodes, reflecting how long it took to receive data on different numbers of stripes. Assuming that receiving 4 out of 5 stripes is sufficient to construct the full data stream (e.g. using redundant coding like MDC/Erasure coding), the 95th percentile join delay is 2.6 seconds.

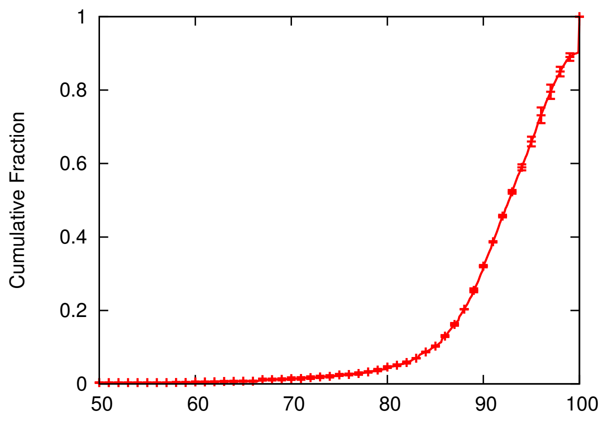
Figure 7(d) shows the CDF of the continuity index among the nodes, calculated with respect to the fraction of data bytes received on all stripes. The average continuity index observed was 99.1. The average node stress (not shown) on the control plane while supporting the multi-tree data overlay is low, with 90% of the nodes handling less than 4 msgs/sec and a maximum node stress of 90 msgs/sec.

We also performed experiments with a higher RI=1.75 and a resulting wider range of node degrees (MIN=1, MAX=6). The results are virtually identical, with a 95th percentile join delay for acquiring 4 out of 5 stripes of 2.2 seconds and an average continuity index of 99.2.

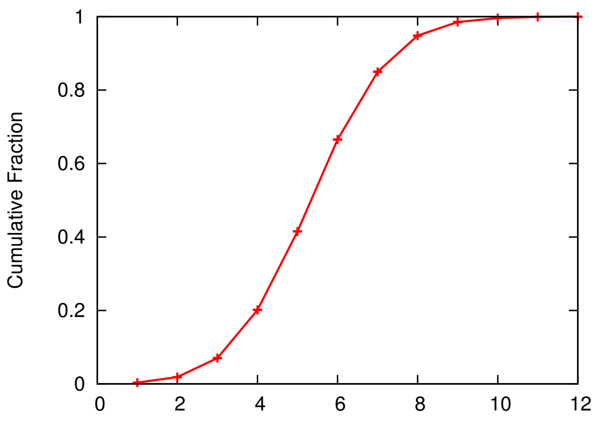
We conclude that SAAR can effectively support a multi-tree data plane design. SAAR can ensure resource balance among the interior-node-disjoint stripe trees in heterogeneous environments. As a result, the resulting CEM system simultaneously realizes the benefits of performance optimized data overlays and the benefits of a multi-tree design in terms of tolerance to loss and membership churn.

5.4 SAAR for Block-based Data Overlays

We built a prototype block-based swarming CEM based on SAAR. The mesh construction, as described in Section 4.3, borrows from Chunkyspread [44]. The swarming block-exchange algorithm we use closely follows Coolstreaming [50]. Briefly, we stripe content into 1 second blocks. Once every second, neighbors exchange their block availability within a sliding window of blocks covering 60 seconds. Missing blocks in this 60 sec buffer



(a) Continuity Index



(b) Block Dissemination Hops

Figure 8: SAAR block-based CEM performance

are requested randomly from the neighbors in inverse proportion to their bandwidth utilizations. We additionally implemented Request-Overriding as explained in Chainsaw [31] to ensure that the multicast source sends out every block at least once.

350 nodes with a heterogeneous degree distribution (mean RI=1.23 and degree caps MIN=1,MAX=2) join the control overlay in the first 10 minutes, and then join the data overlay in the next 5 minutes. They remain in the data overlay for another 10 minutes and then continue to leave/rejoin the data overlay with a mean session time of 2 minutes (exponential distribution) for the remainder of the experiment. We enforce a minimum session time of 60 seconds to allow them to fill their initial buffer worth of 60 secs. To achieve a large instantaneous group size, nodes re-join the same data overlay 10 secs after leaving. Nodes do not depart from the control overlay during the experiment, which ran for approximately one hour.

Figure 8 shows the CDF of the continuity index, and the distribution of overlay hops taken per block in the mesh. The average continuity index is 91.6. The average control node stress (not shown) is low, with 90% of the nodes handling less than 3 msg/sec and a maximum node stress of 80 msgs/sec. Note that the join delay in block-based systems is dominated by the size of the sliding block window, 60 secs in this case.

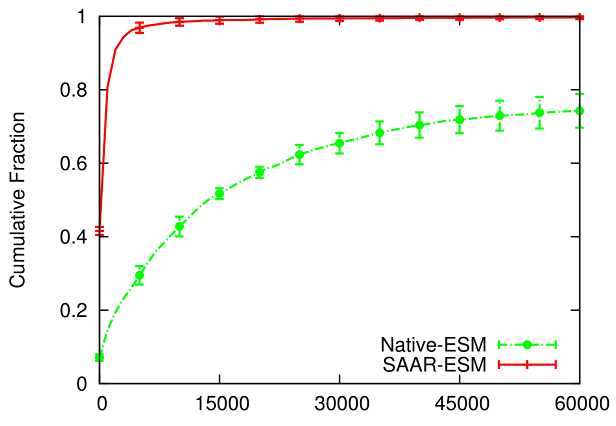
We also performed experiments with degree caps of (MIN=1, MAX=6, RI=1.75). Here, the average continuity index improved to 96.8, while the distribution of overlay hops was similar.

An additional experiment matches a configuration reported in published results for Coolstreaming [50]. 150 nodes with homogeneous node degrees and RI=1.25 are part of the SAAR control overlay and join a data channel within 1 min. At a mean data overlay session time (exponentially distributed) of 50/100/200 sec, we measured an average continuity index of 89, 94 and 98, respectively. The corresponding results reported for Coolstreaming are 89, 91 and 94, respectively. Thus, our implementation appears to perform on par with Coolstreaming. However, differences in the experimental conditions (Modelnet vs. Planetlab, RI=1.25 vs. unspecified RI, 32 Kbps vs. 500 Kbps streaming rate) do not support a stronger conclusion. In summary, our results show that SAAR can support block-based swarming CEMs effectively.

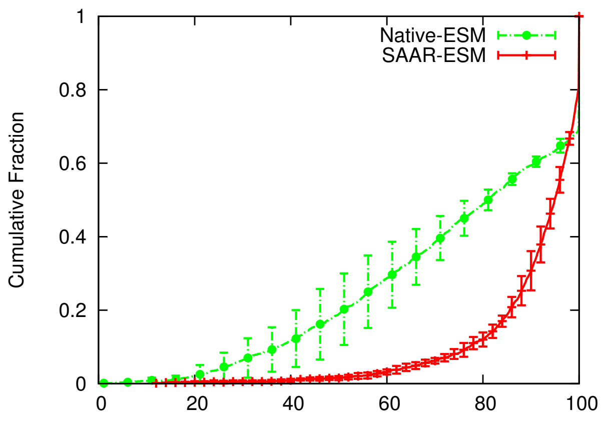
5.5 Planetlab Experiment

To demonstrate that our SAAR-ESM prototype can realize its benefits when deployed in the Internet, we performed an additional experiment in the Planetlab testbed. We use a single-tree CEM with a streaming data rate of 100 Kbps. Approximately 125 nodes (chosen randomly across all continents among nodes that had reasonable load averages) join a channel in the first 2 minutes and then continue to leave/rejoin the channel with a mean session time (exponential distribution) of 2 minutes and a minimum of 15 seconds, for an experiment duration of 15 minutes. The node-degree distribution was heterogeneous and used caps of (MIN=1,MAX=6), RI=1.75 In SAAR-ESM, the nodes are part of the control overlay at the start of the experiment, and they do not depart the control overlay during the experiment. Two group spanning trees were used in SAAR to mitigate the effects of excessive scheduling delays due to high loads on Planetlab machines, which can affect anycast response times.

Figure 9 compares SAAR-ESM and Native-ESM with respect to join delay and continuity index. SAAR-ESM has a 90th percentile join delay and tree repair time (not shown) of 2.5 seconds, which results in good continuity indices. Under high churn, Native-ESM is not able to locate neighbors fast enough. Therefore, it suffers from



(a) Join delay (ms)



(b) Continuity Index

Figure 9: Planetlab single-tree CEM performance

higher join delay and tree repair times, which result in a lower continuity index.

The absolute results obtained with SAAR-ESM on Planetlab are not as good as in Modelnet, although the same trends hold. In absolute terms, the 90th percentile anycast response time in one group spanning tree increased from 500 msec in Modelnet to 3.5 seconds on Planetlab, although the number of anycast hops taken was similar. The continuity indices decreased accordingly. We traced the cause to excessive processing delays on Planetlab nodes, where the 50th and 90th percentile load averages[3] were approximately 10 and 20, respectively. Planetlab is a shared testbed infrastructure that tends to be heavily oversubscribed. We believe that most deployments in the Internet would likely encounter less loaded nodes, and thus achieve results much closer to our Modelnet results.

Native-ESM appears to be less sensitive to the excessive scheduling delays in Planetlab than SAAR-ESM. The likely reason is its proactive epidemic membership protocol, which maintains a list of multiple candidate neighbor nodes at all times. SAAR-ESM could implement an optimization that would have a similar effect: cache the results of previous anycasts and attempt to use nodes on this list while starting a new anycast in parallel. We have not yet implemented this optimization, since we are not convinced it is necessary in most practical deployments.

6 Conclusions

We have presented SAAR, a shared control overlay for CEM systems. SAAR separates the control mechanism from the policy of peer selection in CEM systems. By factoring out the control plane into a separate overlay network, SAAR enables powerful and efficient peer selection, while avoiding constraints on the structure of the data dissemination overlay. Moreover, once decoupled, the control overlay can be shared among many data overlay instances. This sharing increases efficiency and dramatically reduces the delay for joining a channel or switching between channels, which is critical for IPTV.

SAAR's anycast primitive locates appropriate data overlay neighbors based on a constraint and an objective function. The primitive can be used to build and maintain a variety of data overlay organizations. We evaluate a prototype implementation of SAAR experimentally. The results show that SAAR can effectively support single-tree, multi-tree and block-based data plane organizations. Its control efficiency allows it to achieve rapid channel join/switching and high content dissemination quality at low overhead, even under high churn and at large scale.

[3] 5 minute average as reported by 'uptime'

7 Acknowledgments

This work was supported in part by the Max Planck Society, by National Science Foundation grants (ANI-0225660, CNS-0520187, CNS-0085920, CNS-0435382, CNS-0448546, ANI-0092806) and by Bhattacharjee's Slown Fellowship. We would like to thank Jeff Hoye, Andreas Haeberlen and Alan Mislove for their help and advice. We would also like to thank the anonymous reviewers and our shepherd Albert Greenberg for helpful feedback on earlier versions of the paper.

References

[1] Akamai FreeFlow. `http://www.akamai.com`.

[2] S. Banerjee, B. Bhattacharjee, and C. Kommareddy. Scalable application layer multicast. In *Proc. of ACM SIGCOMM*, Aug. 2002.

[3] A. Bharambe, S. Rao, V. Padmanabhan, S. Seshan, and H. Zhang. The impact of heterogeneous bandwidth constraints on DHT-based multicast protocols. In *Proc. of IPTPS '05*, Feb. 2005.

[4] S. Bhattacharjee, M. H. Ammar, E. W. Zegura, V. Shah, and Z. Fei. Application-layer anycasting. In *Proc. of INFOCOM'97*, pages 1388–1396, 1997.

[5] R. Braynard, D. Kostic, A. Rodriguez, J. Chase, and A. Vahdat. Opus: An overlay peer utility service. In *Proc. of 5th International Conference on Open Architectures and Network Programming(OPENARCH '02)*, June 2002.

[6] M. Castro, P. Druschel, Y. Hu, and A. Rowstron. Proximity neighbor selection in tree-based structured peer-to-peer overlays. Technical Report MSR-TR-2003-52, Microsoft Research, 2003.

[7] M. Castro, P. Druschel, A. Kermarrec, A. Nandi, A. Rowstron, and A. Singh. SplitStream: High-bandwidth multicast in a cooperative environment. In *Proc. of SOSP 2003*, Oct. 2003.

[8] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE JSAC*, 20(8), Oct. 2002.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. Rowstron. Scalable application level anycast for highly dynamic groups. In *Proc. NGC'2003*, Sept. 2003.

[10] Y. Chu, A. Ganjam, T. Ng, S. Rao, K. Sripanidkulchai, J. Zhan, and H. Zhang. Early experience with an Internet broadcast system based on overlay multicast. In *Proc. of USENIX Annual Technical Conference*, 2004.

[11] Y. Chu, S. Rao, and H. Zhang. A case for end system multicast. In *ACM Sigmetrics*, pages 1–12, June 2000.

[12] B. Chun, P. Wu, H. Weatherspoon, and J. Kubiatowicz. ChunkCast: An anycast service for large content distribution. In *Proc. of IPTPS '06*, Feb. 2006.

[13] F. Dabek, B. Zhao, P. Druschel, J. Kubiatowicz, and I. Stoica. Towards a common API for structured peer-to-peer overlays. In *Proc. IPTPS '03*, Feb. 2003.

[14] S. Deering and D. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2), May 1990.

[15] J. Dunagan, N. Harvey, M. Jones, M. Theimer, and A. Wolman. Subscriber/volunteer trees: Polite, efficient overlay multicast trees. Technical Report MSR-TR-2004-131, Microsoft Research, 2004.

[16] M. Freedman, E. Freudenthal, and D. Mazieres. Democratizing content publication with Coral. In *Proc. NSDI '04*, Mar. 2004.

[17] M. Freedman, K. Lakshminarayan, and D. Mazieres. Oasis: Anycast for any service. In *Proc. NSDI '06*, May 2006.

[18] Freepastry. http://freepastry.rice.edu/.

[19] A. Ganesh, A. Kermarrec, and L. Massoulie. Scamp: Peer-to-peer lighweight membership service for large-scale group communication. In *Proc. NGC'2001*, Nov. 2001.

[20] V. Goyal. Multiple description coding: Compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, Sept. 2001.

[21] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. ACM SIGCOMM 2003*, Aug. 2003.

[22] K. Gummadi, S.Saroiu, and S. Gribble. King: Estimating latency between arbitrary internet end hosts. In *Proc. ACM SIGCOMM IMW*, Nov. 2002.

[23] X. Hei, C. Liang, J. Liang, Y. Liu, and K. Ross. Insights into PPLive: A measurement study of a large-scale P2P IPTV system. In *Proc. of Workshop on Internet Protocol TV(IPTV) services over World Wide Web(WWW'06)*, May 2006.

[24] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in dynamic network. In *Theory of Computing Systems, Springer verlag*, Mar. 2004.

[25] J. Jannotti, D. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O. Jr. Overcast: Reliable multicasting with an overlay network. In *Proc. of OSDI 2000*, Oct. 2000.

[26] D. Katabi and J. Wroclawski. A framework for scalable global IP-Anycast (GIA). In *Proc. ACM SIGCOMM'00*, 2000.

[27] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of SOSP*, 2003.

[28] A. Nandi, A. Ganjam, P. Druschel, T. Ng, I. Stoica, H. Zhang, and B. Bhattacharjee. SAAR: A shared control plane for overlay multicast. Technical Report Technical Report 2006-2, Max Planck Institute for Software Systems, Oct. 2006.

[29] T. Ng and H. Zhang. A network positioning system for the internet. In *Proc. of USENIX Annual Technical Conference*, 2004.

[30] V. Padmanabhan, H. Wang, P. Chou, and K. Sripanidkulchai. Distributing streaming media content using cooperative networking. In *Proc. of NOSSDAV*, May 2002.

[31] V. Pai, K. Kumar, K. Tamilmani, V. Sambamurthy, and A. Mohr. Chainsaw: Eliminating trees from overlay multicast. In *Proc. of IPTPS 2005*, Feb 2005.

[32] C. Partridge, T. Mendez, and W. Milliken. RFC 1546: Host anycasting service, Nov. 1993.

[33] P. Pietzuch, J. Shneidman, J.Ledlie, M. Welsh, M. Seltzer, and M. Roussopoulos. Evaluating DHT-based service placement for stream-based overlays. In *Proc. IPTPS '05*, Feb. 2005.

[34] Planetlab. http://www.planet-lab.org/.

[35] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM'01*, Aug. 2001.

[36] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Application-level multicast using content-addressable networks. In *Proc. of NGC*, 2001.

[37] R. Renesse, K. Birman, and W.Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management and data mining. *ACM TOCS*, 21(2):164–206, May 2003.

[38] S. Rhea, D.Geels, T. Roscoe, and J. Kubiatowicz. Handling churn in a DHT. In *Proc. of USENIX Annual Technical Conference*, 2004.

[39] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM Middleware 2001*, Nov. 2001.

[40] K. Sripanidkulchai, A. Ganjam, B. Maggs, and H. Zhang. The feasibility of supporting large-scale live streaming applications with dynamic application end-points. In *Proc. of SIGCOMM 2004*, 2004.

[41] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM'2002*, Aug. 2002.

[42] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. ACM SIGCOMM'01*, Aug. 2001.

[43] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proc. of OSDI 2002*, Dec. 2002.

[44] V. Venkataraman, P. Francis, and J. Calandrino. Chunkyspread: Multi-tree unstructured peer-to-peer multicast. In *Proc. IPTPS '06*, Feb. 2006.

[45] V. Vishnumurthy and P. Francis. On heterogeneous overlay construction and random node selection in unstructured p2p networks. In *Proc. INFOCOM 2006*, April 2006.

[46] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. of OSDI 2002*, Dec. 2002.

[47] W. Wang, D. Helder, S. Jamin, and L. Zhang. Overlay optimizations for end-host multicast. In *Proc. of 4th Intl. Workshop on Networked Group Communication (NGC)*, Oct. 2002.

[48] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. ACM SIGCOMM 2004*, Aug. 2004.

[49] B. Zhang, S. Jamin, and L. Zhang. Host multicast: A framework for delivering multicast to end users. In *Proc. of INFOCOM 2002*, 2002.

[50] X. Zhang, J. Liu, B. Li, and T. Yum. Donet: A data-driven overlay network for efficient live media streaming. In *Proc. INFOCOM 2005*, March 2005.

Ricochet: Lateral Error Correction for Time-Critical Multicast

Mahesh Balakrishnan[†], Ken Birman[†], Amar Phanishayee[‡], Stefan Pleisch[†]
[†]*Cornell University and* [‡]*Carnegie Mellon University*
{*mahesh,ken,pleisch*}*@cs.cornell.edu, amarp+@cs.cmu.edu*

Abstract

Ricochet is a low-latency reliable multicast protocol designed for time-critical clustered applications. It uses IP Multicast to transmit data and recovers from packet loss in end-hosts using Lateral Error Correction (LEC), a novel repair mechanism in which XORs are exchanged between receivers and combined across overlapping groups. In datacenters and clusters, application needs frequently dictate large numbers of fine-grained overlapping multicast groups. Existing multicast reliability schemes scale poorly in such settings, providing latency of packet recovery that depends inversely on the data rate within a single group: the lower the data rate, the longer it takes to recover lost packets. LEC is insensitive to the rate of data in any one group and allows each node to split its bandwidth between hundreds to thousands of fine-grained multicast groups without sacrificing timely packet recovery. As a result, Ricochet provides developers with a scalable, reliable and fast multicast primitive to layer under high-level abstractions such as publish-subscribe, group communication and replicated service/object infrastructures. We evaluate Ricochet on a 64-node cluster with up to 1024 groups per node: under various loss rates, it recovers almost all packets using LEC in tens of milliseconds and the remainder with reactive traffic within 200 milliseconds.

1 Introduction

Clusters and datacenters play an increasingly important role in the contemporary computing spectrum, providing back-end computing and storage for a wide range of applications. The modern datacenter is typically composed of hundreds to thousands of inexpensive commodity blade-servers, networked via fast, dedicated interconnects. The software stack running on a single blade-server is a brew of off-the-shelf software: commercial operating systems, proprietary middleware, managed run-time environments and virtual machines, all standardized to reduce complexity and mitigate maintenance costs.

The last decade has seen the migration of time-critical applications to commodity clusters. Application domains ranging from computational finance to air-traffic control and military communication have been driven by scalability and cost concerns to abandon traditional real-time environments for COTS datacenters. In the process, they give up conservative - and arguably unnecessary - guarantees of real-time performance for the promise of massive scalability and multiple nines of timely availability, all at a fraction of the running cost. Delivering on this promise within expanding and increasingly complex datacenters is a non-trivial task, and a wealth of commercial technology has emerged to support clustered applications.

At the heart of commercial datacenter software is *reliable multicast* — used by publish-subscribe and data distribution layers [5, 7] to spread data through clusters at high speeds, by clustered application servers [1, 4, 3] to communicate state, updates and heartbeats between server instances, and by distributed caching infrastructures [2, 6] to rapidly update cached data. The multicast technology used in contemporary industrial products is derivative of protocols developed by academic researchers over the last two decades, aimed at scaling metrics like throughput or latency across dimensions as varied as group size [10, 17], numbers of senders [9], node and network heterogeneity [12], or geographical and routing distance [18, 21]. However, these protocols were primarily designed to extend the reach of multicast to massive networks; they are not optimized for the failure modes of datacenters and may be unstable, inefficient and ineffective when retrofitted to clustered settings. Crucially, they are not designed to cope with the unique scalability demands of time-critical fault-tolerant applications.

We posit that a vital dimension of scalability for clustered applications is the *number of groups* in the system. All the uses of multicast mentioned above induce large numbers of overlapping groups. For example, a computational finance calculator that uses a topic-based pub-sub system to subscribe to a fraction of the equities on the stock market will end up belonging in many multicast groups. Multiple such applications within a datacenter - each subscribing to different sets of equities - can result in arbitrary patterns of group overlap. Similarly, data caching or replication at fine granularity can result in a single node hosting many data items. Replication driven by high-level objectives such as locality, load-balancing or fault-tolerance can lead to distinct overlapping replica sets - and hence, multicast groups - for each item.

In this paper, we propose Ricochet, a time-critical reliable multicast protocol designed to perform well in the multicast patterns induced by clustered applications. Ricochet uses IP Multicast [15] to transmit data and recovers lost packets using *Lateral Error Correction* (LEC), a novel error correction mechanism in which XOR repair packets are probabilistically exchanged between receivers and combined across overlapping multicast groups. The latency of loss recovery in LEC depends inversely on the aggregate rate of data in the system, rather than the rate in any one group. It performs equally well in any arbitrary configuration and cardinality of group overlap, allowing Ricochet to scale to massive numbers of groups while retaining the best characteristics of state-of-the-art multicast technology: even distribution of responsibility among receivers, insensitivity to group size, stable proactive overhead and graceful degradation of performance in the face of increasing loss rates.

1.1 Contributions

- We argue that a critical dimension of scalability for multicast in clustered settings is the number of groups in the system.
- We show that existing reliable multicast protocols have recovery latency characteristics that are inversely dependent on the data rate in a group, and do not perform well when each node is in many low-rate multicast groups.
- We propose Lateral Error Correction, a new reliability mechanism that allows packet recovery latency to be independent of per-group data rate by intelligently combining the repair traffic of multiple groups. We describe the design and implementation of Ricochet, a reliable multicast protocol that uses LEC to achieve massive scalability in the number of groups in the system.
- We extensively evaluate the Ricochet implementation on a 64-node cluster, showing that it performs well with different loss rates, tolerates bursty loss patterns, and is relatively insensitive to grouping patterns and overlaps - providing recovery characteristics that degrade gracefully with the number of groups in the system, as well as other conventional dimensions of scalability.

2 System Model

We consider patterns of multicast usage where each node is in many different groups of small to medium size (10 to 50 nodes). Following the IP Multicast model, a group is defined as a set of receivers for multicast data, and senders do not have to belong to the group to send to it. We expect each node to receive data from a large set of distinct senders, across all the groups it belongs to.

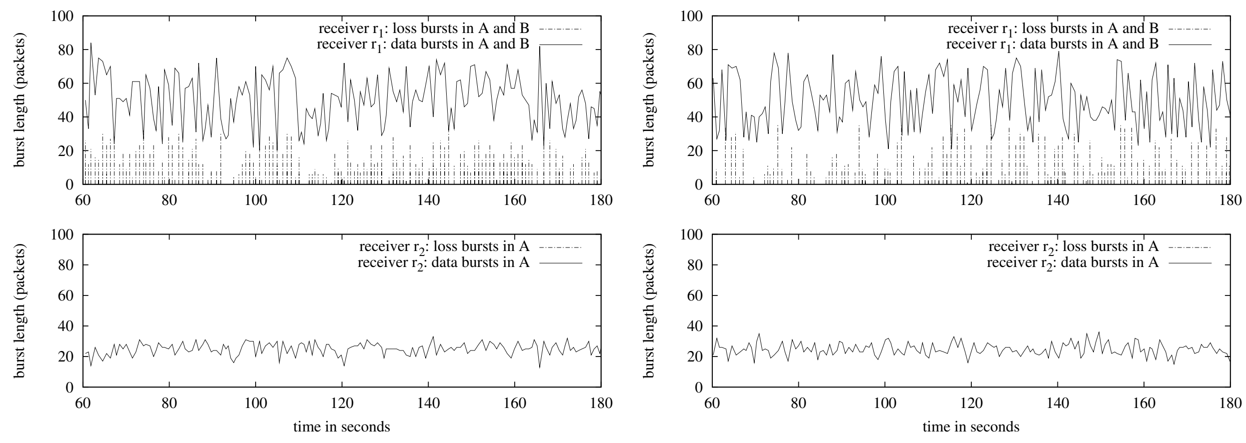
Where does Loss occur in a Datacenter? Datacenter networks have flat routing structures with no more than two or three hops on any end-to-end path. They are typically over-provisioned and of high quality, and packet loss in the network is almost non-existent. In contrast, datacenter end-hosts are inexpensive and easily overloaded; even with high-capacity network interfaces, the commodity OS often drops packets due to buffer overflows caused by traffic spikes or high-priority threads occupying the CPU. Hence, our loss model is one of short packet bursts dropped at the end-host receivers at varying loss rates.

Figure 1 strongly indicates that loss in a datacenter is (a) bursty and (b) independent across end-hosts. In this experiment, a receiver r_1 joins two multicast groups A and B, and another receiver r_2 in the same switching segment joins only group A. From a sender located multiple switches away on the network, we send per-second data bursts of around 25 1KB packets to group A and simultaneously send a burst of 0-50 packets to group B, and measure packet loss at both receivers. We ran this experiment on two networks: a 64-node cluster at Cornell with 1.3 Ghz receivers and the Emulab testbed at Utah with 2 Ghz receivers, all nodes running Linux 2.6.12.

The top graphs in Figure 1 show the traffic bursts and loss bursts at receiver r_1, and the bottom graphs show the same information for r_2. We can see that r_1 gets overloaded and drops packets in bursts of size 1-30 packets, whereas r_2 does not drop any packets — importantly, around 30% of the packets dropped by r_1 are in group A, which is common to both receivers. Hence, loss is both bursty and independent across nodes. Together, these graphs indicate strongly that loss occurs due to buffer overflows at receiver r_1.

The example in Figure 1 is simplistic - each incoming burst of traffic arrives at the receiver within a small number of milliseconds - but conveys a powerful message: it is very easy to trigger significant bursty loss at datacenter end-hosts. The receivers in these experiments were running empty and draining packets continuously out of the kernel, with zero contention for the CPU or the network, whereas the settings of interest to us involve time-critical, possibly CPU-intensive applications running on top of the communication stack.

Further, we expect multi-group settings to intrinsically exhibit bursty incoming traffic of the kind emulated in this experiment — each node in the system receives data from multiple senders in multiple groups and it is likely that the inter-arrival time of data packets at a node will vary widely, even if the traffic rate at one sender or group is steady. In some cases, burstiness of traffic could also occur due to time-critical application behavior - for example, imagine an update in the value of a stock quote triggering off activity in several system components, which then multicast information to a replicated central data-



(a) Cornell 64-node Cluster (b) Utah Emulab Testbed

Figure 1: Datacenter Loss is bursty and uncorrelated across nodes: receiver r_1 (top) joins groups A and B and exhibits bursty loss, whereas receiver r_2 (bottom) joins only group A and experiences zero loss.

store. If we assume that each time-critical component processes the update within a few hundred microseconds, and that inter-node socket-to-socket latency is around fifty microseconds (an actual number from our experimental cluster), the central datastore could easily see a sub-millisecond burst of traffic. In this case, the componentized structure of the application resulted in bursty traffic; in other scenarios, the application domain could be intrinsically prone to bursty input. For example, a financial calculator tracking a set of hundred equities with correlated movements might expect to receive a burst of a hundred packets in multiple groups almost instantaneously.

3 The Design of a Time-Critical Multicast Primitive

In recent years, multicast research has focused almost exclusively on application-level routing mechanisms, or overlay networks ([13] is one example), designed to operate in the wide-area without any existing router support. The need for overlay multicast stems from the lack of IP Multicast coverage in the modern internet, which in turn reflects concerns of administration complexity, scalability, and the risk of multicast 'storms' caused by misbehaving nodes. However, the homogeneity and comparatively limited size of datacenter networks pose few scalability and administration challenges to IP Multicast, making it a viable and attractive option in such settings. In this paper, we restrict ourselves to a more traditional definition of 'reliable multicast', as a reliability layer over IP Multicast. Given that the selection of datacenter hardware is typically influenced by commercial constraints, we believe that any viable solution for this context must be able to run on any mix of existing commodity routers and OS software; hence, we focus exclusively on mechanisms that act at the application-level, ruling out schemes which require router modification, such as PGM [19].

3.1 The Timeliness of (Scalable) Reliable Multicast Protocols

Reliable multicast protocols typically consist of three logical phases: *transmission* of the packet, *discovery* of packet loss, and *recovery* from it. Recovery is a fairly fast operation; once a node knows it is missing a packet, recovering it involves retrieving the packet from some other node. However, in most existing scalable multicast protocols, the time taken to discover packet loss dominates recovery latency heavily in the kind of settings we are interested in. The key insight is that *the discovery latency of reliable multicast protocols is usually inversely dependent on data rate*: for existing protocols, the rate of outgoing data at a single sender in a single group. Existing schemes for reliability in multicast can be roughly divided into the following categories:

ACK/timeout: RMTP [21], RMTP-II [22]. In this approach, receivers send back ACKs (acknowledgements) to the sender of the multicast. This is the trivial extension of unicast reliability to multicast, and is intrinsically unscalable due to ACK implosion; for each sent message, the sender has to process an ACK from every receiver in the group [21]. One work-around is to use ACK aggregation, which allows such solutions to scale in the number of receivers but requires the construction of a tree for every sender to a group. Also, any aggregative mechanism introduces latency, leading to larger time-outs at the sender and delaying loss discovery; hence, ACK trees are unsuitable in time-critical settings.

Gossip-Based: Bimodal Multicast [10], lpbcast [17]. Receivers periodically gossip histories of received packets

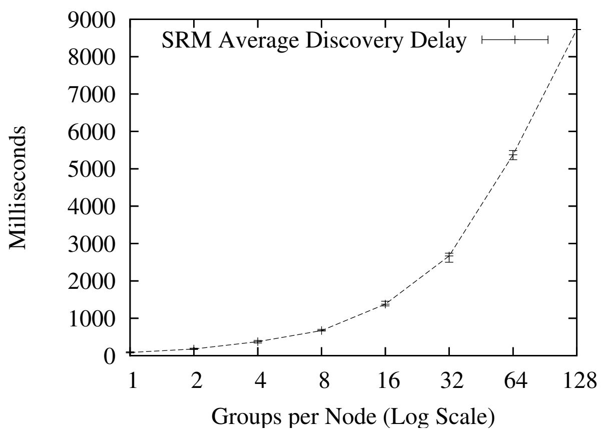


Figure 2: SRM's Discovery Latency vs. Groups per Node, on a 64-node cluster, with groups of 10 nodes each. Error bars are min and max over 10 runs.

with each other. Upon receiving a digest, a receiver compares the contents with its own packet history, sending any packets that are missing from the gossiped history and requesting transmission of any packets missing from its own history. Gossip-based schemes offer scalability in the number of receivers per group, and extreme resilience by diffusing the responsibility of ensuring reliability for each packet over the entire set of receivers. However, they are not designed for time-critical settings: discovery latency is equal to the time period between gossip exchanges (a significant number of milliseconds - 100ms in Bimodal Multicast [10]), and recovery involves a further one or two-phase interaction as the affected node obtains the packet from its gossip contact.

NAK/Sender-based Sequencing: SRM [18]. Senders number outgoing multicasts, and receivers discover packet loss when a subsequent message arrives. Loss discovery latency is thus proportional to the inter-send time at any single sender to a single group - a receiver can't discover a loss in a group until it receives the next packet from the same sender to that group - and consequently depends on the sender's data transmission rate to the group. To illustrate this point, we measured the performance of SRM as we increased the number of groups each node belonged in, keeping the throughput in the system constant by reducing the data rate within each group - as Figure 2 shows, discovery latency of lost packets degrades linearly as each node's bandwidth is increasingly fragmented and each group's rate goes down, increasing the time between two consecutive sends by a sender to the same group. Once discovery occurs in SRM, lost packet recovery is initiated by the receiver, which uses IP multicast (with a suitable TTL value); the sender (or some other receiver), responds with a retransmission, also using IP multicast.

Sender-based FEC [20, 23]: Forward Error Correction schemes involve multicasting redundant error correction information along with data packets, so that receivers can recover lost packets without contacting the sender or any other node. FEC mechanisms involve generating c repair packets for every r data packets, such that any r of the combined set of $r + c$ data and repair packets is sufficient to recover the original r data packets; we term this (r, c) parameter the *rate-of-fire*. FEC mechanisms have the benefit of *tunability*, providing a coherent relationship between overhead and timeliness - the more the number of repair packets generated, the higher the probability of recovering lost packets from the FEC data. Further, FEC based protocols are very stable under stress, since recovery does not induce large degrees of extra traffic. As in NAK protocols, the timeliness of FEC recovery depends on the data transmission rate of a single sender in a single group; the sender can send a repair packet to a group only after sending out r data packets to that group. Fast, efficient encodings such as Tornado codes [11] make sender-based FEC a very attractive option in multicast applications involving a single, dedicated sender; for example, software distribution or internet radio.

Receiver-based FEC [9]: Of the above schemes, ACK-based protocols are intrinsically unsuited for time-critical multi-sender settings, while sender-based sequencing and FEC limit discovery latency to inter-send time at a single sender within a single group. Ideally, we would like discovery latency to be independent of inter-send time, and combine the scalability of a gossip-based scheme with the tunability of FEC. Receiver-based FEC, first introduced in the Slingshot protocol [9], provides such a combination: receivers generate FEC packets from incoming data and exchange these with other randomly chosen receivers. Since FEC packets are generated from *incoming* data at a receiver, the timeliness of loss recovery depends on the rate of data multicast in *the entire group*, rather than the rate at any given sender, allowing scalability in the number of senders to the group.

Slingshot is aimed at single-group settings, recovering from packet loss in time proportional to that group's data rate. Our goal with Ricochet is to achieve recovery latency dependent on the rate of data incoming at a node *across all groups*. Essentially, we want recovery of packets to occur as quickly in a thousand 10 Kbps groups as in a single 10 Mbps group, allowing applications to divide node bandwidth among thousands of multicast groups while maintaining time-critical packet recovery. To achieve this, we introduce Lateral Error Correction, a new form of receiver-generated FEC that probabilistically combines receiver-generated repair traffic across multiple groups to drive down packet recovery latencies.

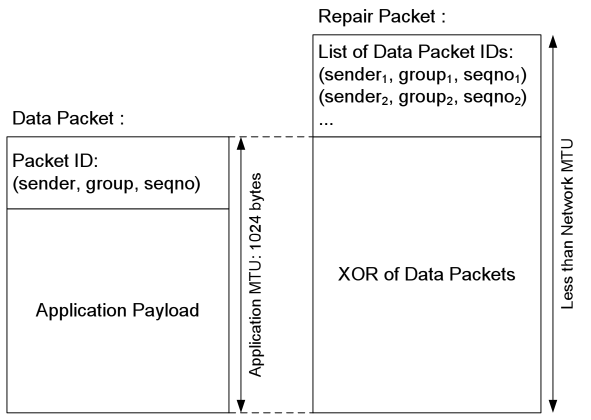


Figure 3: Ricochet Packet Structure

4 Lateral Error Correction and the Ricochet protocol

In Ricochet, each node belongs to a number of groups, and receives data multicast within any of them. The basic operation of the protocol involves generating XORs from incoming data and exchanging them with other randomly selected nodes. Ricochet operates using two different packet types: *data packets* - the actual data multicast within a group - and *repair packets*, which contain recovery information for multiple data packets. Figure 3 shows the structure of these two packet types. Each data packet header contains a packet identifier - a *(sender, group, sequence number)* tuple that identifies it uniquely. A repair packet contains an XOR of multiple data packets, along with a list of their identifiers - we say that the repair packet is *composed* from these data packets, and that the data packets are *included* in the repair packet. An XOR repair composed from r data packets allows recovery of one of them, if all the other $r-1$ data packets are available; the missing data packet is obtained by simply computing the XOR of the repair's payload with the other data packets.

At the core of Ricochet is the LEC engine running at each node that decides on the composition and destinations of repair packets, creating them from incoming data across multiple groups. The operating principle behind LEC is the notion that repair packets sent by a node to another node can be composed from data in any of the multicast groups that are common to them. This allows recovery of lost packets at the receiver of the repair packet to occur within time that's inversely proportional to the aggregate rate of data in all these groups. Figure 4 illustrates this idea: n_1 has groups A and B in common with n_2, and hence it can generate and dispatch repair packets that contain data from both these groups. n_1 needs to wait only until it receives 5 data packets in either A *or* B before it sends a repair packet, allowing faster recovery of lost packets at n_2.

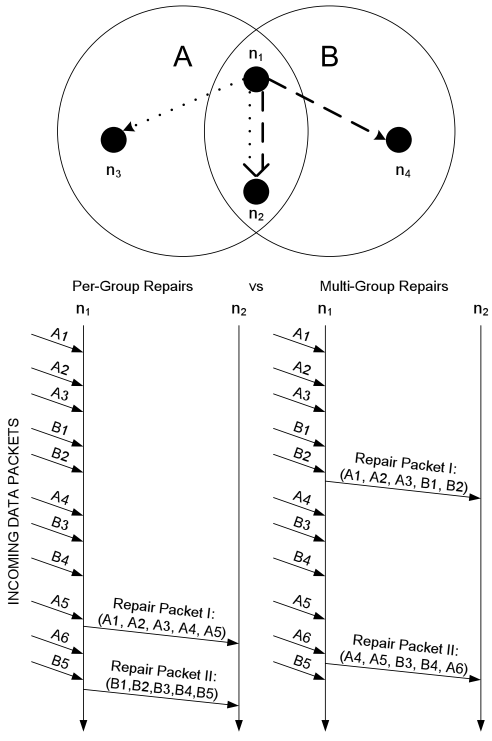


Figure 4: LEC in 2 Groups: Receiver n_1 can send repairs to n_2 that combine data from both groups A and B.

While combining data from different groups in outgoing repair packets drives down recovery time, it tampers with the coherent tunability that single group receiver-based FEC provides. The *rate-of-fire* parameter in receiver-based FEC provides a clear, coherent relationship between overhead and recovery percentage; for every r data packets, c repair packets are generated in the system, resulting in some computable probability of recovering from packet loss. The challenge for LEC is to combine repair traffic for multiple groups while retaining per-group overhead and recovery percentages, so that each individual group can maintain its own rate-of-fire. To do so, we abstract out the essential properties of receiver-based FEC that we wish to maintain:

1. Coherent, Tunable Per-Group Overhead: For every data packet that a node receives in a group with rate-of-fire (r, c), it sends out *an average of* c *repair packets* including that data packet to other nodes in the group.
2. Randomness: Destination nodes for repair packets are picked *randomly*, with no node receiving more or less repairs than any other node, on average.

LEC supports overlapping groups with the same r component and different c values in their rate-of-fire parameter. In LEC, the rate-of-fire parameter is translated into the following guarantee: For every data packet d that a node receives in a group with rate-of-fire (r, c), it selects an average of c nodes from the group randomly and sends each

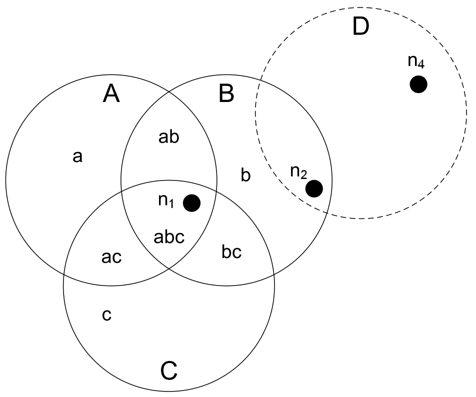


Figure 5: n_1 belongs to groups A, B, C: it divides them into disjoint regions abc, ab, ac, bc, a, b, c

of these nodes exactly one repair packet that includes d. In other words, the node sends an average of c repair packets containing d to the group. In the following section, we describe the algorithm that LEC uses to compose and dispatch repair packets while maintaining this guarantee.

4.1 Algorithm Overview

Ricochet is a symmetric protocol - exactly the same LEC algorithm and supporting code runs at every node - and hence, we can describe its operation from the vantage point of a single node, n_1.

4.1.1 Regions

The LEC engine running at n_1 divides n_1's neighborhood - the set of nodes it shares one or more multicast groups with - into *regions*, and uses this information to construct and disseminate repair packets. Regions are simply the disjoint intersections of all the groups that n_1 belongs to. Figure 5 shows the regions in a hypothetical system, where n_1 is in three groups, A, B and C. We denote groups by upper-case letters and regions by the concatenation of the group names in lowercase; i.e, abc is a region formed by the intersection of A, B and C. In our example, the neighborhood set of n_1 is carved into seven regions: abc, ac, ab, bc, a, b and c, essentially the power set of the set of groups involved. Readers may be alarmed that this transformation results in an exponential number of regions, but this is not the case; we are only concerned with non-empty intersections, the cardinality of which is bounded by the number of nodes in the system, as each node belongs to exactly one intersection (see Section 4.1.4). Note that n_1 does not belong to group D and is oblivious to it; it observes n_2 as belonging to region b, rather than bd, and is not aware of n_4's existence.

4.1.2 Selecting targets from regions, not groups

Instead of selecting targets for repairs randomly from the entire group, LEC selects targets randomly from *each region*. The number of targets selected from a region is set such that:

1. It is proportional to the size of the region
2. The total number of targets selected, across regions, is equal to the c value of the group

Hence, for a given group A with rate-of-fire (r, c_A), the number of targets selected by LEC in a particular region, say abc, is equal to $c_A * \frac{|abc|}{|A|}$, where $|x|$ is the number of nodes in the region or group x. We denote the number of targets selected by LEC in region abc for packets in group A as c_A^{abc}. Figure 6 shows n_1 selecting targets for repairs from the regions of A.

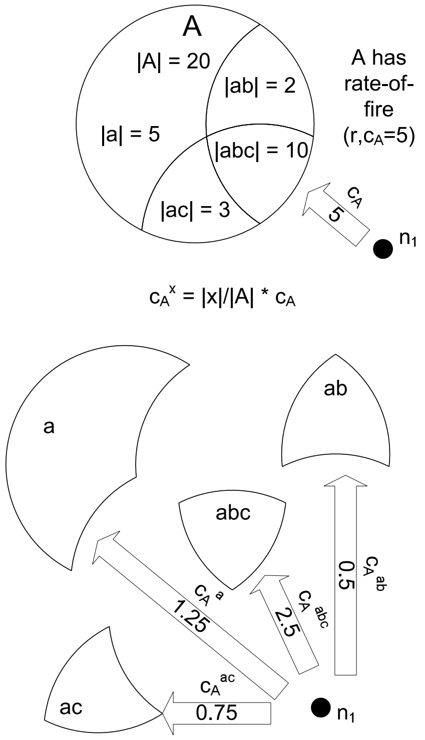


Figure 6: n_1 selects proportionally sized chunks of c_A from the regions of A

Note that LEC may pick a different number of targets from a region for packets in a different group; for example, c_A^{abc} differs from c_B^{abc}. Selecting targets in this manner also preserves randomness of selection; if we rephrase the task of target selection as a sampling problem, where a random sample of size c has to be selected from the group, selecting targets from regions corresponds to *stratified sampling* [14], a technique from statistical theory.

4.1.3 Why select targets from regions?

Selecting targets from regions instead of groups allows LEC to construct repair packets from multiple groups; since we know that all nodes in region ab are interested in data from groups A and B, we can create composite

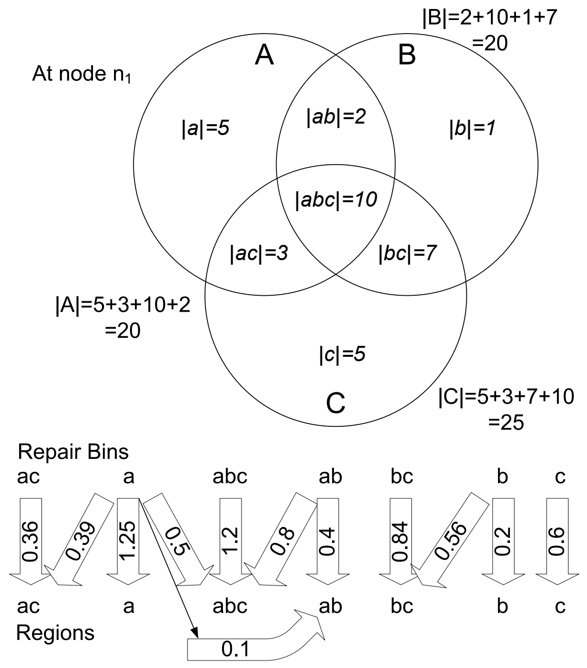


Figure 7: Mappings between repair bins and regions: the repair bin for ab selects 0.4 targets from region ab and 0.8 from abc for every repair packet. Here, $c_A = 5$, $c_B = 4$, and $c_C = 3$.

repair packets from incoming data packets in both groups and send them to nodes in that region.

Single-group receiver-based FEC [9] is implemented using a simple construct called a *repair bin*, which collects incoming data within the group. When a repair bin reaches a threshold size of r, a repair packet is generated from its contents and sent to c randomly selected nodes in the group, after which the bin is cleared. Extending the repair bin construct to regions seems simple; a bin can be maintained for each region, collecting data packets received in any of the groups composing that region. When the bin fills up to size r, it can generate a repair packet containing data from all these groups, and send it to targets selected from within the region.

Using per-region repair bins raises an interesting question: if we construct a composite repair packet from data in groups A, B, and C, how many targets should we select from region abc for this repair packet - c_A^{abc}, c_B^{abc}, or c_C^{abc}? One possible solution is to pick the maximum of these values. If $c_A^{abc} \geq c_B^{abc} \geq c_C^{abc}$, then we would select c_A^{abc}. However, a data packet in group B, when added to the repair bin for the region abc would be sent to an average of c_A^{abc} targets in the region; resulting in more repair packets containing that data packet sent to the region than required (c_B^{abc}), which results in more repair packets sent to the entire group. Hence, more overhead is expended per data packet in group B than required by its (r, c_B) value; a similar argument holds for data packets in group C as well.

Algorithm 1 Algorithm for Setting Up Repair Bins

```
1:  Code at node $n_i$:

2:  upon Change in Group Membership do
3:      while L not empty                {L is the list of regions}
        do
4:          Select and remove the region $R_i = abc...z$ from
            $L$ with highest number of groups involved (break
            ties in any order)
5:          Set $R_t = R_i$
6:          while $R_t \neq \epsilon$ do
7:              set $c_{min}$ to $min(c_A^{R_t}, c_B^{R_t}...)$, where {A,B,...}
                is the set of groups forming $R_t$
8:              Set number of targets selected by $R_i$'s repair
                bin from region $R_t$ to $c_{min}$
9:              Remove $G$ from $R_t$, for all groups $G$ where
                $c_G^{R_t} = c_{min}$
10:             For each remaining group $G'$ in $R_t$, set $c_{G'}^{R_t} =$
                $c_{G'}^{R_t} - c_{min}$
```

Instead, we choose the *minimum* of values; this, as expected, results in a lower level of overhead for groups A and B than required, resulting in a lower fraction of packets recovered from LEC. To rectify this we send the additional compensating repair packets to the region abc from the repair bins for regions a and b. The repair bin for region a would select $c_A^{abc} - c_C^{abc}$ destinations, on an average, for every repair packet it generates; this is in addition to the c_A^a destinations it selects from region a.

A more sophisticated version of the above strategy involves iteratively obtaining the required repair packets from regions involving the remaining groups; for instance, we would have the repair bin for ab select the minimum of c_A^{abc} and c_B^{abc} - which happens to be c_B^{abc} - from abc, and then have the repair bin for a select the remainder value, $c_A^{abc} - c_B^{abc}$, from abc. Algorithm 1 illustrates the final approach adopted by LEC, and Figure 7 shows the output of this algorithm for an example scenario. A repair bin selects a non-integral number of nodes from an intersection by alternating between its floor and ceiling probabilistically, in order to maintain the average at that number.

4.1.4 Complexity

The algorithm described above is run every time nodes join or leave any of the multicast groups that n_1 is part of. The algorithm has complexity $O(I \cdot d)$, where I is the number of populated regions (i.e, with one or more nodes in them), and d is the maximum number of groups that form a region. Note that I at n_1 is bounded from above by the cardinality of the set of nodes that share a multicast group with n_1, since regions are disjoint and each node exists in exactly one of them. d is bounded by the number of groups that n_1 belongs to.

4.2 Implementation Details

Our implementation of Ricochet is in Java. Below, we discuss the details of the implementation, along with the performance optimizations involved - some obvious and others subtle.

4.2.1 Repair Bins

A Ricochet repair bin is a lightweight structure holding an XOR and a list of data packets, and supporting an add operation that takes in a data packet and includes it in the internal state. The repair bin is associated with a particular region, receiving all data packets incoming in any of the groups forming that region. It has a list of regions from which it selects targets for repair packets; each of these regions is associated with a value, which is the average number of targets which must be selected from that region for an outgoing repair packet. In most cases, as shown in Figure 7, the value associated with a region is not an integer; as mentioned before, the repair bin alternates between the floor and the ceiling of the value to maintain the average at the value itself. For example, in Figure 7, the repair bin for abc has to select 1.2 targets from abc, on average; hence, it generates a random number between 0 and 1 for each outgoing repair packet, selecting 1 node if the random number is more than 0.2, and 2 nodes otherwise.

4.2.2 Staggering for Bursty Loss

A crucial algorithmic optimization in Ricochet is *staggering* - also known as interleaving [23] - which provides resilience to bursty loss. Given a sequence of data packets to encode, a stagger of 2 would entail constructing one repair packet from the 1st, 3rd, 5th... packets, and another repair packet from the 2nd, 4th, 6th... packets. The stagger value defines the number of repairs simultaneously being constructed, as well as the distance in the sequence between two data packets included in the same repair packet. Consequently, a stagger of i allows us to tolerate a loss burst of size i while resulting in a proportional slowdown in recovery latency, since we now have to wait for $O(i*r)$ data packets before despatching repair packets.

In conventional sender-based FEC, staggering is not a very attractive option, providing tolerance to very small bursts at the cost of multiplying the already prohibitive loss discovery latency. However, LEC recovers packets so quickly that we can tolerate a slowdown of a factor of ten without leaving the tens of milliseconds range; additionally, a small stagger at the sender allows us to tolerate very large bursts of lost packets at the receiver, especially since the burst is dissipated among multiple groups and senders. Ricochet implements a stagger of i by the simple expedient of duplicating each logical repair bin into i instances; when a data packet is added to the logical repair bin, it is actually added to a particular instance of the repair bin, chosen in round-robin fashion. Instances of a duplicated repair bin behave exactly as single repair bins do, generating repair packets and sending them to regions when they get filled up.

4.2.3 Multi-Group Views

Each Ricochet node has a *multi-group view*, which contains membership information about other nodes in the system that share one or more multicast groups with it. In traditional group communication literature, a *view* is simply a list of members in a single group [24]; in contrast, a Ricochet node's multi-group view divides the groups that it belongs to into a number of regions, and contains a list of members lying in each region. Ricochet uses the multi-group view at a node to determine the sizes of regions and groups, to set up repair bins using the LEC algorithm. Also, the per-region lists in the multi-view are used to select destinations for repair packets. The multi-group view at n_1 - and consequently the group and intersection sizes - does not include n_1 itself.

4.2.4 Membership and Failure Detection

Ricochet can plug into any existing membership and failure detection infrastructure, as long as it is provided with reasonably up-to-date views of per-group membership by some external service. In our implementation, we use simple versions of Group Membership (GMS) and Failure Detection (FD) services, which execute on high-end server machines. If the GMS receives a notification from the FD that a node has failed, or it receives a join/leave to a group from a node, it sends an update to all nodes in the affected group(s). The GMS is not aware of regions; it maintains conventional per-group lists of nodes, and sends per-group updates when membership changes. For example, if node n_{55} joins group A, the update sent by the GMS to every node in A would be a 3-tuple: *(Join, A,* n_{55}*)*. Individual nodes process these updates to construct multi-group views relative to their own membership.

Since the GMS does not maintain region data, it has to scale only in the number of groups in the system; this can be easily done by partitioning the service on group id and running each partition on a different server. For instance, one machine is responsible for groups A and B, another for C and D, and so on. Similarly, the FD can be partitioned on a topological criterion; one machine on each rack is responsible for monitoring other nodes on the rack by pinging them periodically. For fault-tolerance, each partition of the GMS can be replicated on multiple machines using a strongly consistent protocol like Paxos. The FD can have a hierarchical structure to recover from failures; a smaller set of machines ping the per-rack failure detectors, and each other in a chain. We believe that such a semi-centralized solution is appropriate and sufficient in a datacenter setting, where connectivity and membership are typically stable. Crucially, the protocol itself does not need consistent membership, and degrades gracefully with the degree of inconsistency in the views; if a failed node is included in a view, performance will dip fractionally in all the groups it belongs to as the repairs sent to it by other nodes are wasted.

4.2.5 Performance

Since Ricochet creates LEC information from each incoming data packet, the critical communication path that a data packet follows within the protocol is vital in determining eventual recovery times and the maximum sustainable throughput. XORs are computed in each repair bin incrementally, as packets are added to the bin. A crucial optimization used is pre-computation of the number of destinations that the repair bin sends out a repair to, across all the regions that it sends repairs to: Instead of constructing a repair and deciding on the number of destinations once the bin fills up, the repair bin precomputes this number and constructs the repair only if the number is greater than 0. When the bin overflows and clears itself, the expected number of destinations for the next repair packet is generated. This restricts the average number of two-input XORs per data packet to c (from the rate-of-fire) in the worst case - which occurs when no single repair bin selects more than 1 destination, and hence each outgoing repair packet is a unique XOR.

4.2.6 Buffering and Loss Control

LEC - like any other form of FEC - works best when losses are not in concentrated bursts. Ricochet maintains an application-level buffer with the aim of minimizing in-kernel losses, serviced by a separate thread that continuously drains packets from the kernel. If memory at end-hosts is constrained and the application-level buffer is bounded, we use customized packet-drop policies to handle overflows: a randomly selected packet from the buffer is dropped and the new packet is accommodated instead. In practice, this results in a sequence of almost random losses from the buffer, which are easy to recover using FEC traffic. Whether the application-level buffer is bounded or not, it ensures that packet losses in the kernel are reduced to short bursts that occur only during periods of overload or CPU contention. We evaluate Ricochet against loss bursts of up to 100 packets, though in practice we expect the kind of loss pattern shown in 1, where few bursts are greater than 20-30 packets, even with highly concentrated traffic spikes.

4.2.7 NAK Layer for 100% Recovery

Ricochet recovers a high percentage of lost packets via the proactive LEC traffic; for certain applications, this probabilistic guarantee of packet recovery is sufficient and even desirable in cases where data 'expires' and there is no utility in recovering it after a certain number of milliseconds. However, the majority of applications require 100% recovery of lost data, and Ricochet uses a reactive NAK layer to provide this guarantee. If a receiver does not recover a packet through LEC traffic within a timeout period after discovery of loss, it sends an explicit NAK to the sender and requests a retransmission. While this NAK layer does result in extra reactive repair traffic, two factors separate it from traditional NAK mechanisms: firstly, recovery can potentially occur very quickly - within a few hundred milliseconds - since for almost all lost packets discovery of loss takes place within milliseconds through LEC traffic. Secondly, the NAK layer is meant solely as a backup mechanism for LEC and responsible for recovering a very small percentage of total loss, and hence the extra overhead is minimal.

4.2.8 Optimizations

Ricochet maintains a buffer of unusable repair packets that enable it to utilize incoming repair packets better. If one repair packet is missing exactly one more data packet than another repair packet, and both are missing at least one data packet, Ricochet obtains the extra data packet by XORing the two repair packets. Also, it maintains a list of unusable repair packets which is checked intermittently to see if recent data packet recoveries and receives have made any old repair packets usable.

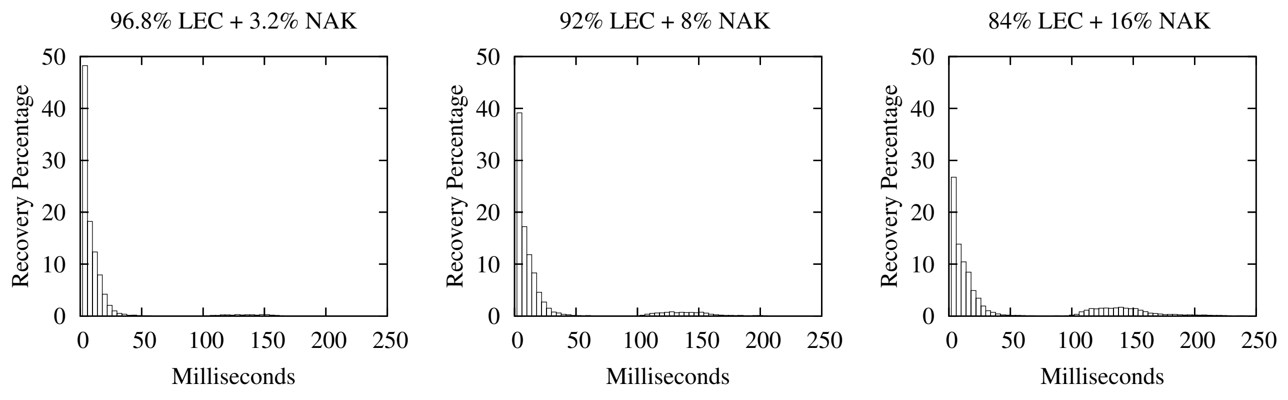
4.2.9 Message Ordering

As presented, Ricochet provides multicast reliability but does not deliver messages in the same order at all receivers. We are primarily concerned with building an extremely rapid multicast primitive that can be used by applications that require unordered reliable delivery as well as layered under ordering protocols with stronger delivery properties. For instance, Ricochet can be used as a reliable transport by any of the existing mechanisms for total ordering [16] — in separate work [8], we describe one such technique that predicts out-of-order delivery in datacenters to optimize ordering delays.

5 Evaluation

We evaluated our Java implementation of Ricochet on a 64-node cluster, comprising of four racks of 16 nodes each, interconnected via two levels of switches. Each node has a single 1.3 GHz CPU with 512 Mb RAM, runs Linux 2.6.12 and has two 100 Mbps network interfaces, one for control and the other for experimental traffic. Typical socket-to-socket latency within the cluster is around 50 microseconds. In the following experiments, for a given loss rate L, three different loss models are used:

· **uniform** - also known as the Bernoulli model [25] - refers to dropping packets with uniform probability equal to the loss rate L.



(a) 10% Loss Rate **(b) 15% Loss Rate** **(c) 20% Loss Rate**

Figure 8: Distribution of Recoveries: LEC + NAK for varying degrees of loss

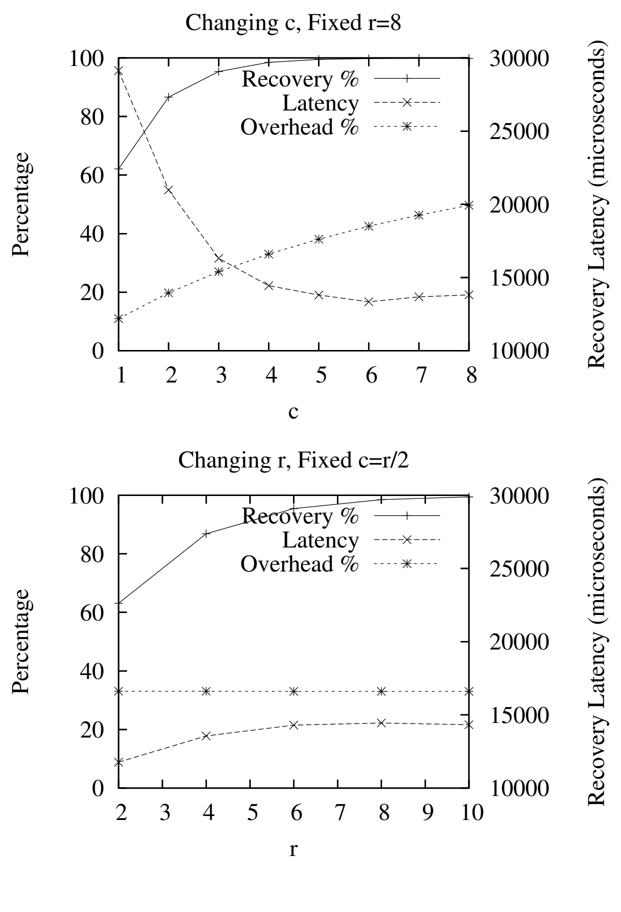


Figure 9: Tuning LEC : tradeoff points available between recovery %, overhead % (left y-axis) and avg recovery latency (right y-axis) by changing the rate-of-fire (r, c).

· **bursty** involves dropping packets in equal bursts of length b. The probability of starting a loss burst is set so that each burst is of exactly b packets and the loss rate is maintained at L. This is not a realistic model but allows us to precisely measure performance relative to specific burst lengths.

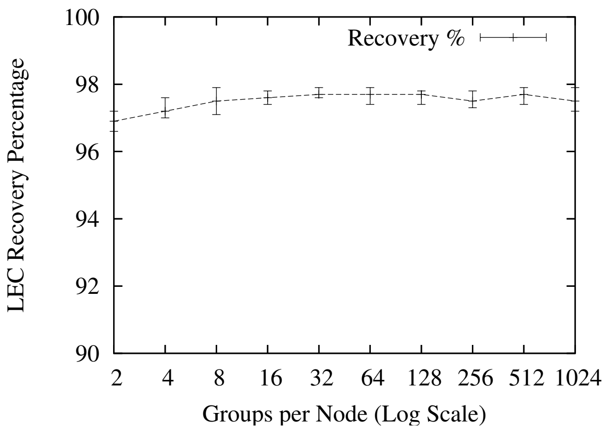
· **markov** drops packets using a simple 2-state markov chain, where each node alternates between a lossy and a lossless state, and the probabilities are set so that the average length of a loss burst is m and the loss rate is L, as described in [25].

In experiments with multiple groups, nodes are assigned to groups at random, and the following formula is used to relate the variables in the grouping pattern: $n * d = g * s$, where n is the number of nodes in the system (64 in most of the experiments), d is the degree of membership, i.e. the number of groups each node joins, g is the total number of groups in the system, and s is the average size of each group. For example, in a 16-node setting where each node joins 512 groups and each group is of size 8, g is set to $\frac{16*512}{8} \approx 1024$. Each node is then assigned to 512 randomly picked groups out of 1024. Hence, the grouping patterns for each experiment is completely represented by a (n, d, s) tuple.

For every run, we set the sending rate at a node such that the total system rate of incoming messages is 64000 packets per second, or 1000 packets per node per second. Data packets are 1K bytes in size. Each point in the following graphs - other than Figure 8, which shows distributions for single runs - is an average of 5 runs. A run lasts 30 seconds and produces $\approx$ 2 million receive events in the system.

5.1 Distribution of Recoveries in Ricochet

First, we provide a snapshot of what typical packet recovery timelines look like in Ricochet. Earlier, we made



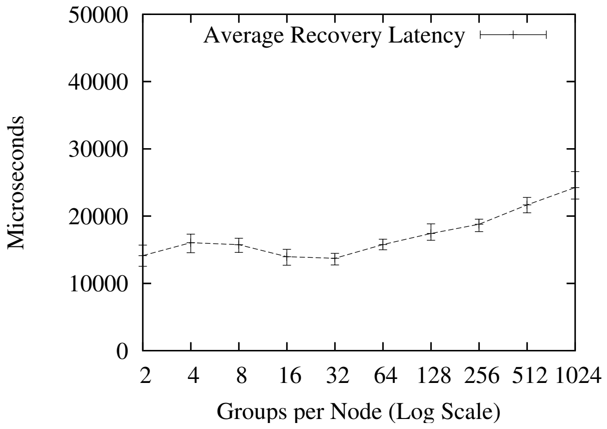
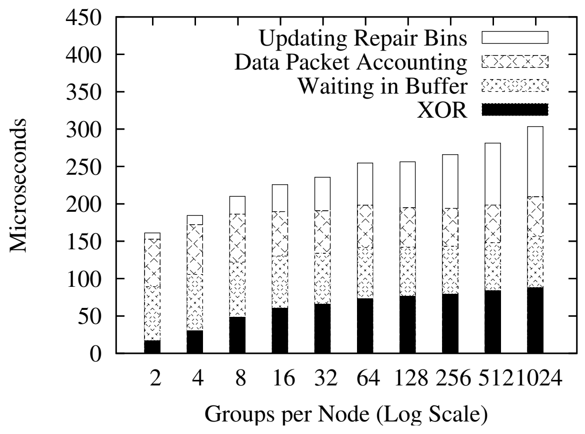


Figure 10: Scalability in Groups



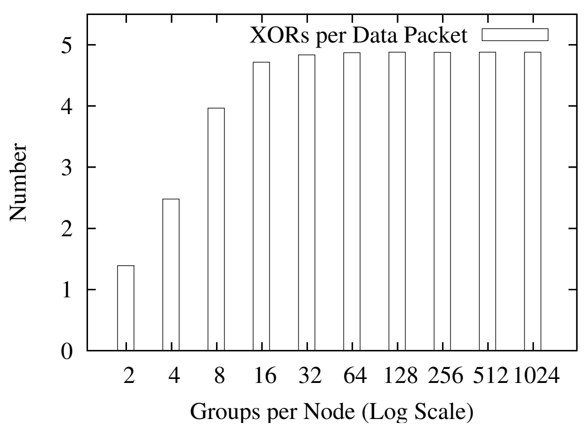


Figure 11: CPU time and XORs per data packet

the assertion that Ricochet discovers the loss of almost all packets very quickly through LEC traffic, recovers a majority of these instantly and recovers the remainder using an optional NAK layer. In Figure 8, we show the histogram of packet recovery latencies for a 16-node run with degree of membership $d = 128$ and group size $s = 10$. We use a simplistic NAK layer that starts unicasting NAKs to the original sender of the multicast 100 milliseconds after discovery of loss, and retries at 50 millisecond intervals. Figure 8 shows three scenarios: under uniform loss rates of 10%, 15%, and 20%, different fractions of packet loss are recovered through LEC and the remainder via reactive NAKs. These graphs illustrate the meaning of the LEC recovery percentage: if this number is high, more packets are recovered very quickly without extra traffic in the initial segment of the graphs, and less reactive overhead is induced by the NAK layer. Importantly, even with a recovery percentage as low as 84% in Figure 8(c), we are able to recover all packets within 250 milliseconds with a crude NAK layer due to early LEC-based discovery of loss. For the remaining experiments, we will switch the NAK layer off and focus solely on LEC performance; also, since we found this distribution of recovery latencies to be fairly representative, we present only the percentage of lost packets recovered using LEC and the average latency of these recoveries. Experiment Setup: $(n = 16, d = 128, s = 10)$, Loss Model: Uniform, [10%, 15%, 20%].

5.2 Tunability of LEC in multiple groups

The Slingshot protocol [9] illustrated the tunability of receiver-generated FEC for a single group; we include a similar graph for Ricochet in Figure 9, showing that the rate-of-fire parameter (r, c) provides a knob to tune LEC's recovery characteristics. In Figure 9.a, we can see that increasing the c value for constant $r = 8$ increases the recovery percentage and lowers recovery latency by expending more overhead - measured as the percentage of repair packets to all packets. In Figure 9.b, we see the impact of increasing r, keeping the ratio of c to r - and consequently, the overhead - constant. For the rest of the experiments, we set the rate-of-fire at $(r = 8, c = 5)$. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Uniform, 1%.

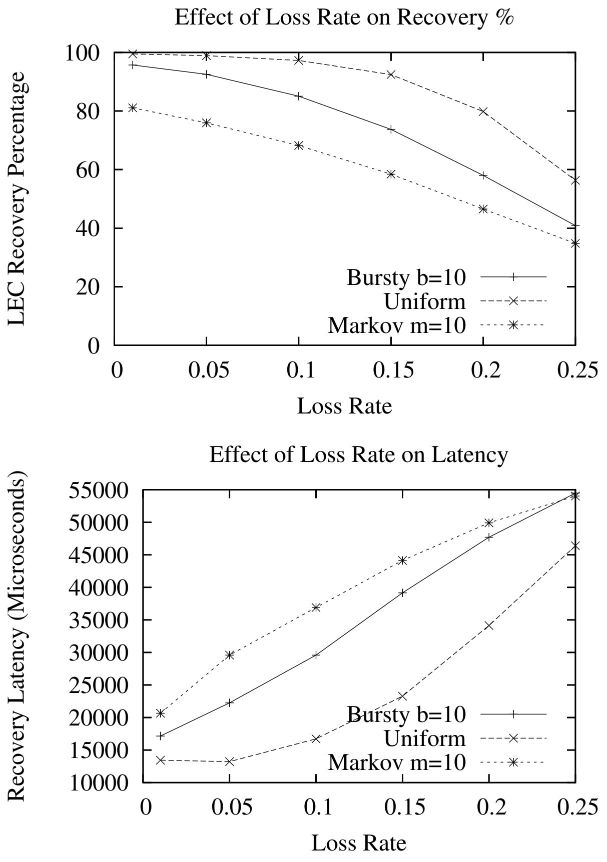


Figure 12: Impact of Loss Rate on LEC

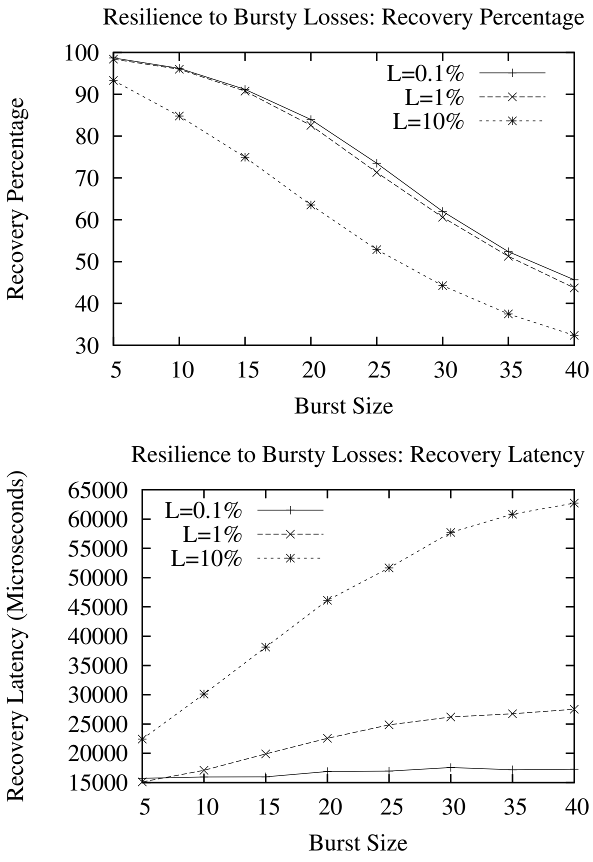


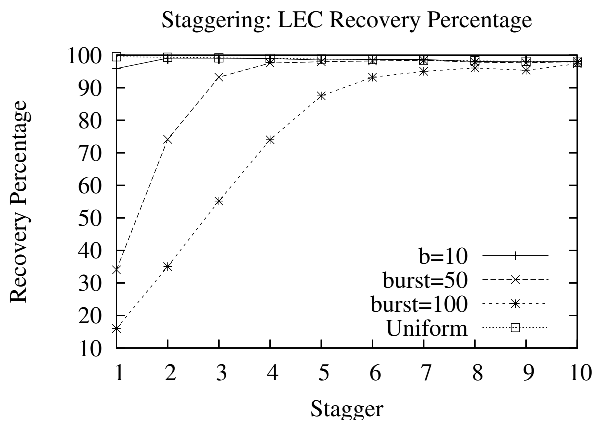
Figure 13: Resilience to Burstiness

5.3 Scalability

Next, we examine the scalability of Ricochet to large numbers of groups. Figure 10 shows that increasing the degree of membership for each node from 2 to 1024 has almost no effect on the percentage of packets recovered via LEC, and causes a slow increase in average recovery latency. The x-axis in these graphs is log-scale, and hence a straight line increase is actually logarithmic with respect to the number of groups and represents excellent scalability. The increase in recovery latency towards the right side of the graph is due to Ricochet having to deal internally with the representation of large numbers of groups; we examine this phenomenon later in this section.

For a comparison point, we refer readers back to SRM's discovery latency in Figure 2: in 128 groups, SRM discovery took place at 9 seconds. In our experiments, SRM recovery took place roughly 4 seconds after discovery in all cases. While fine-tuning the SRM implementation for clustered settings should eliminate that 4 second gap between discovery and recovery, at 128 groups Ricochet surpasses SRM's best possible recovery performance of 5 seconds by between 2 and 3 orders of magnitude.

Though Ricochet's recovery characteristics scale well in the number of groups, it is important that the computational overhead imposed by the protocol on nodes stays manageable, given that time-critical applications are expected to run over it. Figure 11 shows the scalability of an important metric: the time taken to process a single data packet. The straight line increase against a log x-axis shows that per-packet processing time increases logarithmically with the number of groups - doubling the number of groups results in a constant increase in processing time. The increase in processing time towards the latter half of the graph is due to the increase in the number of repair bins with the number of groups. While we considered 1024 groups adequate scalability, Ricochet can potentially scale to more groups with further optimization, such as creating bins only for occupied regions. In the current implementation, per-packet processing time goes from 160 microseconds for 2 groups to 300 microseconds for 1024, supporting throughput exceeding a thousand packets per second. Figure 11 also shows the average number of XORs per incoming data packet. As stated in section 4.2.2, the number of XORs stays under 5 - the value of c from the rate-of-fire (r, c). Experiment Setup: $(n = 64, d = *, s = 10)$, Loss Model: Uniform, 1%.



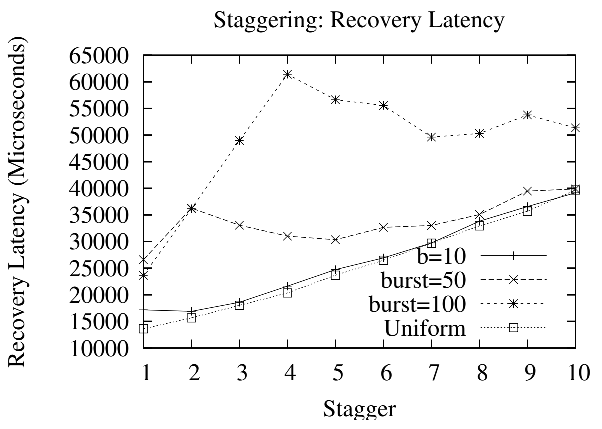


Figure 14: Staggering allows Ricochet to recover from long bursts of loss.

5.4 Loss Rate and LEC Effectiveness

Figure 12 shows the impact of the Loss Rate on LEC recovery characteristics, under the three loss models. Both LEC recovery percentages and latencies degrade gracefully: with an unrealistically high loss rate of 25%, Ricochet still recovers 40% of lost packets at an average of 60 milliseconds. For uniform and bursty loss models, recovery percentage stays above 90% with a 5% loss rate; markov does not fare as well, even at 1% loss rate, primarily because it induces bursts much longer than its average of 10 - the max burst in this setting averages at 50 packets. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: *.

5.5 Resilience to Bursty Losses

As we noted before, a major criticism of FEC schemes is their fragility in the face of bursty packet loss. Figure 13 shows that Ricochet is naturally resilient to small loss bursts, without the stagger optimization - however, as the burst size increases, the percentage of packets recovered using LEC degrades substantially. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty.

However, switching on the stagger optimization described in Section 4.2.2 increases Ricochet's resilience to burstiness tremendously, without impacting recovery latency much. Figure 14 shows that setting an appropriate stagger value allows Ricochet to handle large bursts of loss: for a burst size as large as 100, a stagger of 6 enables recovery of more than 90% lost packets at an average latency of around 50 milliseconds. Experiment Setup: $(n = 64, d = 128, s = 10)$, Loss Model: Bursty, 1%.

5.6 Effect of Group and System Size

What happens to LEC performance when the average group size in the cluster is large compared to the total number of nodes? Figure 15 shows that recovery percentages are almost unaffected, staying above 99% in this scenario, but recovery latency is impacted by more than a factor of 2 as we triple group size from 16 to 48 in a 64-node setting. Note that this measures the impact of the size of the group relative to the entire system; receiver-based FEC has been shown to scale well in a single isolated group to hundreds of nodes [9]. Experiment Setup: $(n = 64, d = 128, s = *)$, Loss Model: Uniform, 1%.

While we could not evaluate to system sizes beyond 64 nodes, Ricochet should be oblivious to the size of the entire system, since each node is only concerned with the groups it belongs to. We ran 4 instances of Ricochet on each node to obtain an emulated 256 node system with each instance in 128 groups, and the resulting recovery percentage of 98% - albeit with a degraded average recovery latency of nearly 200 milliseconds due to network and CPU contention - confirmed our intuition of the protocol's fundamental insensitivity to system size.

6 Future Work

One avenue of research involves embedding more complex error codes such as Tornado [11] in LEC; however, the use of XOR has significant implications for the design of the algorithm, and using a different encoding might require significant changes. LEC uses XOR for its simplicity and speed, and as our evaluation showed, we obtain properties on par with more sophisticated encodings, including tunability and burst resilience. We plan on replacing our simplistic NAK layer with a version optimized for bulk transfer, providing an efficient backup for LEC when sustained bursts occur of hundreds of packets or more. Another line of work involves making the parameters for LEC - such as rate-of-fire and stagger - adaptive, reacting to meet varying load and network characteristics. We are currently working with industry partners to layer Ricochet under data distribution, publish-subscribe and web-service interfaces, as well as building protocols with stronger ordering and atomicity properties over it.

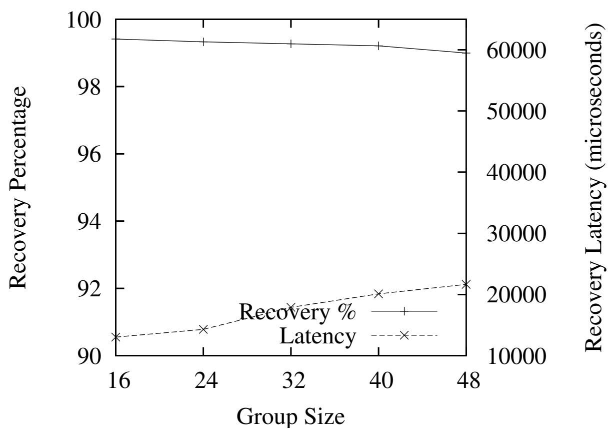


Figure 15: Effect of Group Size

7 Conclusion

We believe that the next generation of time-critical applications will execute on commodity clusters, using the techniques of massive redundancy, fault-tolerance and scalable communication currently available to distributed systems practitioners. Such applications will require a multicast primitive that delivers data at the speed of hardware multicast in failure-free operation and recovers from packet loss within milliseconds irrespective of the pattern of usage. Ricochet provides applications with massive scalability in multiple dimensions - crucially, it scales in the number of groups in the system, performing well under arbitrary grouping patterns and overlaps. A clustered communication primitive with good timing properties can ultimately be of use to applications in diverse domains not normally considered time-critical - e-tailers, online webservers and enterprise applications, to name a few.

Acknowledgments

We received invaluable comments from Dave Andersen, Danny Dolev, Tudor Marian, Art Munson, Robbert van Renesse, Emin Gun Sirer, Niraj Tolia and Einar Vollset. We would like to thank our shepherd Mike Dahlin, as well as all the anonymous reviewers of the paper.

References

[1] Bea weblogic. `http://www.bea.com/framework.jsp?CNT=index.htm&FP=/content/products/weblogic`, 2006.

[2] Gemstone gemfire. `http://www.gemstone.com/products/gemfire/enterprise.php`, 2006.

[3] Ibm websphere. `www.ibm.com/software/webservers/appserv/was/`, 2006.

[4] Jboss. `http://labs.jboss.com/portal/`, 2006.

[5] Real-time innovations data distribution service. `http://www.rti.com/products/data_distribution/index.html`, 2006.

[6] Tangosol coherence. `http://www.tangosol.com/html/coherence-overview.shtml`, 2006.

[7] Tibco rendezvous. `http://www.tibco.com/software/messaging/rendezvous.jsp`, 2006.

[8] M. Balakrishnan, K. Birman, and A. Phanishayee. Plato: Predictive latency-aware total ordering. In *IEEE SRDS*, 2006.

[9] M. Balakrishnan, S. Pleisch, and K. Birman. Slingshot: Time-critical multicast for clustered applications. In *IEEE Network Computing and Applications*, 2005.

[10] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, 1999.

[11] J. W. Byers, M. Luby, M. Mitzenmacher, and A. Rege. A digital fountain approach to reliable distribution of bulk data. In *ACM SIGCOMM '98 Conference*, pages 56–67, New York, NY, USA, 1998. ACM Press.

[12] Y. Chawathe, S. McCanne, and E. A. Brewer. Rmx: Reliable multicast for heterogeneous networks. In *INFOCOM*, pages 795–804, 2000.

[13] Y. Chu, S. Rao, S. Seshan, and H. Zhang. Enabling conferencing applications on the internet using an overlay muilticast architecture. In *ACM SIGCOMM*, pages 55–67, New York, NY, USA, 2001. ACM Press.

[14] W. G. Cochran. *Sampling Techniques, 3rd Edition.* John Wiley, 1977.

[15] S. E. Deering and D. R. Cheriton. Multicast routing in datagram internetworks and extended lans. *ACM Trans. Comput. Syst.*, 8(2):85–110, 1990.

[16] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[17] P. T. Eugster, R. Guerraoui, S. B. Handurukande, P. Kouznetsov, and A.-M. Kermarrec. Lightweight probabilistic broadcast. *ACM Trans. Comput. Syst.*, 21(4):341–374, 2003.

[18] S. Floyd, V. Jacobson, C.-G. Liu, S. McCanne, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. *IEEE/ACM Trans. Netw.*, 5(6):784–803, 1997.

[19] J. Gemmel, T. Montgomery, T. Speakman, N. Bhaskar, and J. Crowcroft. The pgm reliable multicast protocol. *IEEE Network*, 17(1):16–22, Jan 2003.

[20] C. Huitema. The case for packet level fec. In *PfHSN '96: Proceedings of the TC6 WG6.1/6.4 Fifth International Workshop on Protocols for High-Speed Networks V*, pages 109–120, London, UK, UK, 1997. Chapman & Hall, Ltd.

[21] J. C. Lin and S. Paul. RMTP: A reliable multicast transport protocol. In *INFOCOM*, pages 1414–1424, San Francisco, CA, Mar. 1996.

[22] T. Montgomery, B. Whetten, M. Basavaiah, S. Paul, N. Rastogi, J. Conlan, and T. Yeh. The RMTP-II protocol, Apr. 1998. IETF Internet Draft.

[23] J. Nonnenmacher, E. Biersack, and D. Towsley. Parity-based loss recovery for reliable multicast transmission. In *Proceedings of the ACM SIGCOMM '97 conference*, pages 289–300, New York, NY, USA, 1997. ACM Press.

[24] P. Verissimo and L. Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[25] M. Yajnik, S. B. Moon, J. F. Kurose, and D. F. Towsley. Measurement and modeling of the temporal dependence in packet loss. In *INFOCOM*, pages 345–352, 1999.

WiLDNet: Design and Implementation of High Performance WiFi Based Long Distance Networks [*]

Rabin Patra[†] [‡] Sergiu Nedevschi[†‡] Sonesh Surana[†] Anmol Sheth[§]
Lakshminarayanan Subramanian[¶] Eric Brewer[†‡]

Abstract

WiFi-based Long Distance (WiLD) networks with links as long as 50–100 km have the potential to provide connectivity at substantially lower costs than traditional approaches. However, real-world deployments of such networks yield very poor end-to-end performance. First, the current 802.11 MAC protocol has fundamental shortcomings when used over long distances. Second, WiLD networks can exhibit high and variable loss characteristics, thereby severely limiting end-to-end throughput.

This paper describes the design, implementation and evaluation of WiLDNet, a system that overcomes these two problems and provides enhanced end-to-end performance in WiLD networks. To address the protocol shortcomings, WiLDNet makes several essential changes to the 802.11 MAC protocol, but continues to exploit standard (low-cost) WiFi network cards. To better handle losses and improve link utilization, WiLDNet uses an adaptive loss-recovery mechanism using FEC and bulk acknowledgments. Based on a real-world deployment, WiLDNet provides a 2–5 fold improvement in TCP/UDP throughput (along with significantly reduced loss rates) in comparison to the best throughput achievable by conventional 802.11. WiLDNet can also be configured to adapt to a range of end-to-end performance requirements (bandwidth, delay, loss).

1 Introduction

Many developing regions around the world, especially in rural or remote areas, require low-cost network connectivity solutions. Traditional approaches based on telephone, cellular, satellite or fiber have proved to be an expensive proposition especially in low population density and low-income regions. In Africa, even when cellular or satellite coverage is available in rural regions, bandwidth is extremely expensive (e.g. satellite bandwidth is about US$3000/Mbps per month) [15]. Cellular and WiMax [25], another proposed solution, require a minimum user density to amortize the cost of the basestation that is so far too high for rural areas. Finally, all of these solutions focus on licensed spectrum and carrier-based deployment, which limits their usefulness to the kind of "grass roots" projects typical for developing regions.

WiFi-based Long Distance (WiLD) networks [8, 9, 23] are emerging as a low-cost connectivity solution and are increasingly being deployed in developing regions. The primary cost gains arise from the use of low-cost and low-power single-board computers and high-volume low-cost off-the-shelf 802.11 wireless cards using unlicensed spectrum. The nodes are also lightweight and don't need expensive towers [6]. These networks are very different from the short-range multi-hop urban mesh networks [5]. Unlike mesh networks, which use omnidirectional antennas to cater to short ranges (less than 1–2 km at most), WiLD networks are comprised of point-to-point wireless links that use high-gain directional antennas (e.g. 24 dBi, 8° beam-width) with line of sight (LOS) over long distances (10–100 km).

Despite the promise of WiLD networks as a low-cost network connectivity solution, the real-world deployments of such networks face many challenges [23]. Our experience has shown that in particular, the performance of WiLD networks in real-world deployments is abysmal. There are two main reasons for this poor performance. First, the stock 802.11 protocol has fundamental *protocol shortcomings* that make it ill-suited for WiLD environments. Three specific shortcomings include: (a) the *802.11 link-level recovery* mechanism results in low utilization; (b) at *long distances frequent collisions occur because of the failure of CSMA/CA*; (c) WiLD networks experience *inter-link interference* which introduces the need for synchronizing packet transmissions at each node [17]. The second problem is that the links in our WiLD network deployments (in US, India, Ghana) experienced very *high and variable packet loss rates* induced by external factors (primarily external WiFi interference in our deployment); under such high loss conditions, TCP flows hardly progress and continuously experience timeouts.

In this paper, we describe the design and implementation of WiLDNet, a system that addresses all the aforementioned problems and provides enhanced end-to-end performance in multi-hop WiLD networks. Prior to our study, the only work addressing this problem was

[*]This work was partly supported by National Science Foundation grant No. 0326582 and Intel Research.

[†]University of California, Berkeley
[‡]Intel Research, Berkeley
[§]University of Colorado, Boulder
[¶]New York University

2P [17], a MAC protocol proposed by Raman *et al*. The 2P design primarily addresses inter-link interference, and proposes a TDMA-style protocol with synchronous node transmissions. The design of WiLDNet leverages and builds on top of 2P, making additional changes to further improve link utilization and to make the system robust to packet loss. The key factors that distinguish WiLDNet from 2P and the stock 802.11 protocol are:

1. Improving link utilization using bulk acknowledgments: The current 802.11 protocol uses a stop-and-wait link recovery mechanism, which when used over long distances with high round-trip times leads to under-utilization of the channel. To improve link utilization, WiLDNet uses a bulk packet acknowledgment protocol.

2. Designing TDMA in lossy environments: The stock 802.11 CSMA/CA mechanism is inappropriate for WiLD settings since it cannot assess the state of the channel at the receiver. 2P proposed a basic TDMA mechanism (instead of CSMA/CA) that explicitly synchronized transmissions at each node to prevent inter-link interference. However, with high packet loss rates, explicit synchronization can lead to deadlock scenarios due to loss of synchronization marker packets. In WiLDNet, we use an implicit approach, using loose time synchronization among nodes to determine a TDMA schedule that is not affected by packet loss.

3. Handling high packet loss rates: In our WiLD network deployments, we found that external WiFi interference is the primary source of packet loss. The emergence of many WiFi deployments, even in developing regions, will exacerbate this problem. In WiLDNet, we use an adaptive loss-recovery mechanism that uses a combination of FEC and bulk acknowledgments to significantly reduce the perceived loss rate and to increase the end-to-end throughput. We show that WiLDNet's link-layer recovery mechanism is much more efficient than a higher-layer recovery mechanisms such as Snoop [2].

4. Application-based parameter configuration: Different applications have varying requirements in terms of bandwidth, loss, delay and jitter. In WiLDNet, configuring the TDMA and recovery parameters (time slot period, FEC, number of retries) provides a tradeoff spectrum across different end-to-end properties. We explore these tradeoffs and show that WiLDNet can be configured to suit a wide range of goals.

We have implemented all our modifications as a *shim layer* above the driver using the Click modular router [11]. We have deployed WiLDNet in our campus testbed of 6 long-distance wireless links. Figure 1 shows the topology of our campus testbed. Apart from the design and implementation of WiLDNet, we have had two years experience in deploying and maintaining two production WiLD networks in India and Ghana that support real users. Our network at the Aravind Eye Hospital, India, provides interactive patient-doctor video-conferencing services between the hospital and five surrounding villages (10–25 km away from the hospital). It is currently being used for about 2000 remote patient examinations per month. The design of WiLDNet that is presented in this paper has continuously evolved in the past two years to solve many of the performance problems that we faced in our deployments.

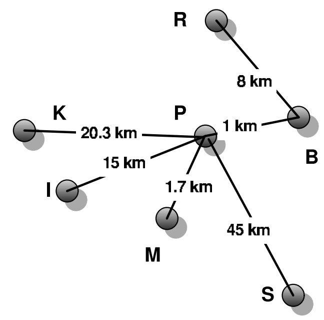


Figure 1: Overview of the WiLD campus testbed (not to scale)

Using a detailed performance evaluation, we roughly observe a 2–5 fold improvement in the TCP throughput over WiLDNet in comparison to the best achievable TCP throughput obtained by making minor driver changes to the standard 802.11 MAC across a wide variety of settings. On our outdoor testbed, we get upto 5 Mbps of TCP throughput over 3 hops under lossy channel conditions, which is 2.5 times more than that of standard 802.11b. The bandwidth overhead of our loss-recovery mechanisms is minimal. In the near future we intend to transition our system from the campus testbed into the two production networks in India and Ghana.

2 WiLD Performance Issues

In this section, we describe in detail two important causes for poor end-to-end performance in WiLD networks: (a) 802.11 protocol shortcomings; (b) high and variable loss-rates in the underlying channel induced by external factors. We begin by providing a brief description of WiLD networks in Section 2.1. In Section 2.3, we elaborate on three protocol shortcomings of 802.11 in WiLD settings: (a) inefficient link-level recovery; (b) collisions at long distances and (c) inter-link interference. For each of these, we show that just manipulating driver level parameters is insufficient to achieve good performance over long-distance links. Then in Section 2.4, we summarize the results of our study of the loss characteristics in our deployed WiLD networks. We observed the primary cause of these losses to be external WiFi interference and not multi-path effects. Finally, in Section 2.5, we discuss the effect of these two causes on TCP performance.

2.1 WiLD Networks: An Introduction

The IEEE 802.11 standard (WiFi) was designed for wireless broadcast environments with many hosts in close vicinity competing for channel access. Wireless radios are half-duplex and cannot listen while transmitting; consequently, a CSMA/CA (carrier-sense multiple-access/collision avoidance) mechanism is used to reduce collisions. Unlike standard WiFi networks, WiFi-based Long Distance (WiLD) networks use multi-hop point-to-point links, where each link can be as long as 100 km. To achieve long distances in single point-to-point links, nodes use directional antennas with gains as high as 30dBi, and may use high-power wireless cards with up to 400mW of transmit power. Additionally, in multi-hop settings, nodes have multiple radios with one radio per fixed point-to-point link to each neighbor. Each radio can operate on different channels if required. This is different from standard 802.11 networks where nodes route traffic through an access point and contend for the medium on a single channel. Some real life deployments of WiLD networks include the Akshaya network [24], the Digital Gangetic Plains project [4], and the CRCnet project [8]. The Akshaya network is one of the largest wireless deployments in the world with over 400 nodes and links going up to 30 km.

2.2 Experimental Setup

We use three different experimental setups to conduct measurements and to evaluate WiLDNet.

Campus testbed: Figure 1 is our real-world campus testbed on which we have currently deployed WiLDNet. The campus testbed consists of links ranging from 1 to 45 km, with end points located in areas with varying levels of external WiFi interference. We also use one of the links in our Ghana network (65km).

Wireless Channel Emulator: The channel emulator (Spirent 5500 [21]) enables repeatable experiments by keeping the link conditions stable for the duration of the experiment. Moreover, by introducing specific propagation delays we can emulate very long links and hence study the effect of long propagation delays. We can also study this in isolation of external interference by placing the end host radios in RF isolation boxes.

Indoor multi-hop testbed: We perform controlled multi-hop experiments on an indoor multi-hop testbed consisting of 4 nodes placed in RF isolated boxes. The setup was designed to recreate conditions similar to long outdoor links where transmissions from local radios interfere with each other but simultaneous reception on multiple local radio interfaces is possible. We can also control the amount of external interference by placing an additional wireless node in each isolation box just to transmit packets mimicking a real interferer. The amount of interference is controlled by the rate of the CBR traffic sent by this node. The indoor setup features very small propagation delay on the links; we use it only to perform experiments evaluating TDMA scheduling and loss recovery from interference.

We use Atheros 802.11 a/b/g radios for all our experiments. The wireless nodes are 266 MHz x86 Geode single board computers running Linux 2.4.26. The choice of this hardware platform is motivated by the low cost ($140) and the low power consumption ($<$ 5W). We use *iperf* to measure UDP and TCP throughput. The madwifi Atheros driver was modified to collect relevant PHY and MAC layer information.

2.3 802.11 Protocol Shortcomings

In this section, we study the three main limitations of the 802.11 protocol: the inefficient link-layer recovery mechanism, collisions in long-distance links, and inter-link interference. These limitations make 802.11 ill-suited even in the case of a single WiLD link. Based on extensive experiments, we also show that modifying the driver-level parameters of 802.11 is insufficient to achieve good performance.

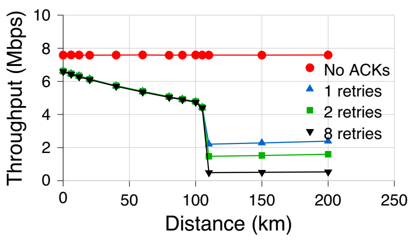
2.3.1 Inefficient Link-Layer Recovery

The 802.11 MAC uses a simple stop-and-wait protocol, with each packet independently acknowledged. Upon successfully receiving a packet, the receiver node is required to send an acknowledgment within a tight time bound (ACKTimeout), or the sender has to retransmit. This mechanism has two drawbacks:

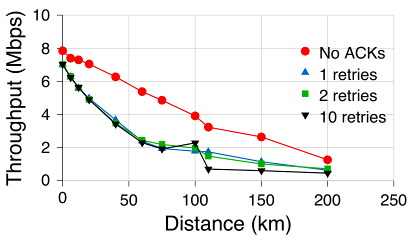
• As the link distance increases, propagation delay increases as well, and the sender waits for a longer time for the ACK to return. This decreases channel utilization.

• If the time it takes for the ACK to return exceeds the ACKTimeout parameter, the sender will retransmit unnecessarily and waste bandwidth.

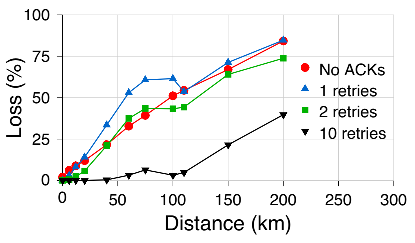
We illustrate these problems by performing experiments using the wireless channel emulator. To emulate long distances, we configure the emulator to introduce a delay to emulate links ranging from 0–200 km. Figure 2(a) shows the performance of the 802.11 stop-and-wait link recovery mechanism over increasing link distances. With the MAC-layer ACKs turned off (No ACKs), we achieve a throughput of 7.6 Mbps at the PHY layer data rate of 11 Mbps. When MAC ACKs are enabled, we adjust the ACK timeout as the distance increases. In this case, the sender waits for an ACK after each transmission, and we observe decreasing channel utilization as the propagation delay increases. At 110 km, the propagation delay exceeds the maximum ACK timeout (746μs for Atheros, but smaller and fixed for Prism 2.5 chipsets) and the sender always times out before the ACKs can arrive. We notice a sharp decrease in received throughput, as the sender retries to send the packet repeatedly (even though the packets were most likely received), until the maximum number of retries is reached (this happens because, if an ACK is late, it is ignored). This causes the received throughput to stabilize at $BW_{110km}/(no\_of\_retries + 1)$.



(a) Unidirectional UDP throughput



(b) Bidirectional UDP throughput



(c) Bidirectional UDP loss

Figure 2: UDP throughputs for standard 802.11 CSMA on single emulated link. ACK timeouts were adjusted with increasing distance (on Atheros cards). Traffic is 1440 byte CBR UDP packets in 802.11b at PHY layer datarate of 11Mbps.

2.3.2 Collisions on long-distance links

The 802.11 protocol uses a CSMA/CA channel-access mechanism, in which nodes listen to the medium for a specified time period (DIFS) before transmitting a packet, thus ensuring that the channel is idle before transmission. This translates to a maximum allowable distance at which collisions can be avoided of about 15km for 802.11b (DIFS is 50μs), 10.2 kms for 802.11a and 8.4km for 802.11g. For longer links it is possible for a node to start transmitting a packet unaware of another packet transmission at the other end. As the propagation delay increases, this probability of loss due to collisions increases.

We illustrate the above-mentioned effect by using a simple experiment: we send bidirectional UDP traffic at the maximum possible sending rate on the emulated link and measure the percentage of packets successfully received at each end. Figure 2(c) shows how the packet loss rate increases with distance. Figure 2(b) shows the sum of the throughputs achieved at both ends for bidirectional UDP traffic as we increase the distance for a link. Note that there are no losses due to attenuation or outside interference in this controlled experiment; all of the losses are due to collisions.

A possible solution to this issue would be to increase the DIFS time interval in order to permit longer propagation delays. However, just as in the case of the ACK timeout, this approach would decrease channel utilization substantially for longer links. Furthermore, we are not aware of any 802.11 chipsets that allow the DIFS interval to be configured.

2.3.3 Multiple Link Interference

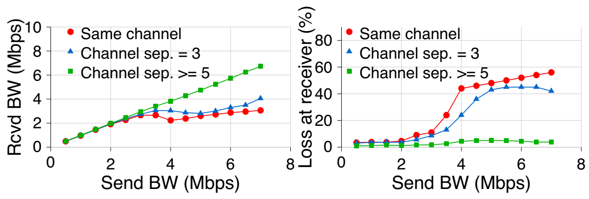
Another important source of errors is the interference between adjacent 802.11 links operating in the same channel or in overlapping channels. Although interference between adjacent links can be avoided by using non-overlapping channels, there are numerous reasons that make it advantageous to operate adjacent links on the same frequency channel, as described by Raman *et al*. [17]. Moreover, there are WiLD topologies such as the Akshaya network [24] where different channels cannot be allocated to all the pairs of adjacent links, given the high connectivity degree of several nodes.

Inter-link interference occurs because the high-power radios create a strong RF field in the vicinity of the radio, enough to interfere with the receptions at nearby radios. Directional antennas also have sufficiently high gain (4–8 dBi) side lobes [4] in addition to the main lobes.

The first type of problem occurs when multiple radios attached to the same node attempt to transmit at the same time. As soon as one radio starts transmitting after sensing the carrier to be idle, all other radios in the vicinity find the carrier to be busy and backoff. This is desirable in a broadcast network to avoid collisions between two senders at any receiver node. However, in our network where each of these radios transmits over point-to-point long distance links to independent receivers, this backoff leads to suboptimal throughput. A second problem occurs when packets being received at one link collide with packets simultaneously transmitted on some other link on the same node. The signal strength of packets transmitted locally on a node overwhelms any packet reception on other local radios.

In order to illustrate these effects, we perform experiments on the real-world setup presented in Figure 1. First, we attempt to simultaneously transmit UDP packets to both K and M from node P. The total send throughput on both links is 14.20 Mbps when they are on non-overlapping channels (separation $\geq$ 4) but drops to only 7.88 Mbps when on the same channel. Next we send UDP packets from node M to node K, relayed through node P at different transmitting rates. We then measure received throughput and packet loss rate for various channel spacing between the two adjacent links, as presented in Figures 3(a) and 3(b). We observe that interference does reduce the utilization of the individual links and significantly increases the link loss rate (even in the case of partially overlapping channels).

Therefore, the maximum channel diversity that one can simultaneously use at a single node in the case of 802.11(b) is restricted to 3 (channels 1,6,11) which may not be sufficient for many WiLD networks. This motivates the need for a scheme that allows the efficient



(a) UDP throughput received (b) UDP loss at receiver

Figure 3: Effect of interference on received UDP throughput and error rate when sending from M to K through a relay node, P. Channel separation is no. of channels in 802.11b. Traffic is 1440 byte CBR UDP packets in 802.11b at PHY layer datarate of 11Mbps.

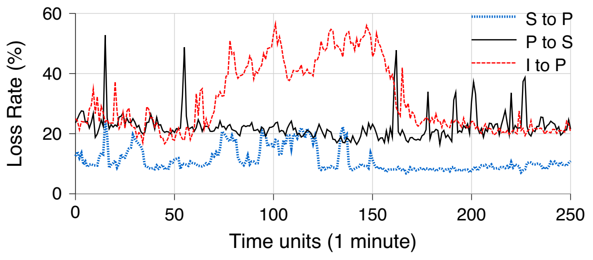


Figure 4: Packet loss variation on 2 links over a period of about 4 hours. Traffic was 1Mbps CBR UDP packets of 1440 bytes each at a PHY datarate of 11Mbps in 802.11b.

operation of same-channel adjacent links. This can be achieved by using a mechanism similar to the one used in 2P [17], that synchronizes both packet transmission and reception across adjacent links to avoid interference and improve throughput.

2.4 Channel Induced Loss

Apart from protocol shortcomings, another cause for poor performance is high packet loss rates in the underlying channel due to external factors. We refer to these as *channel induced losses*. In this section, we briefly summarize the relevant conclusions from our study (Sheth *et al.* [20]) where we conduct a detailed analysis of loss characterization on our WiLD network deployments.

Loss magnitude and variability: Figure 4 illustrates the loss variation across time on two different links in our testbed. We find that the loss is highly varying with time and there are bursts of high loss of lengths varying from few milliseconds up to several minutes. However on the urban links, there is always a non-zero residual that varies between 1–20%. The residual loss rates in our rural links are negligible. Finally, we found the loss characteristics along a single link to be highly asymmetric. One example is illustrated in Figure 4 where we observe that average loss rate from S to P was lower (10%) than the loss from P to S (20%).

Sources of loss: Our study (Sheth *et al.* [20]) investigates two potential sources for channel losses on WiLD links: external WiFi interference and multipath interference. It finds external WiFi interference to be the dominant source of packet loss and multipath to have a much smaller effect.

Multipath has a small effect because the delay spreads in WiLD environments are an order of magnitude lower than those in mesh networks. This is because as link distances increase, the path delay difference between the primary line-of-sight (LOS) path and secondary reflected paths becomes small enough to avoid inter-symbol interference (ISI). On the other hand, the primary path signal can be significantly attenuated from the secondary paths that undergo a phase shift of 180° after reflection. This is verified by our measurements [20] where we see that all our long-distance links in rural areas have very low loss. In comparison, an urban mesh network deployment (like Roofnet) has many short, non line-of-sight links and thus loss from ISI is a much bigger problem.

However if WiLD links are deployed in the presence of external interfering sources, the hidden terminal problem can be much worse than in the case of an urban mesh network (with omnidirectional antennas). Due to the highly directional nature of the transmission, a larger fraction of interfering sources within range of the receiver act as hidden terminals since they cannot sense the sender's transmissions. In addition, due to long propagation delays, even sources within the range of a directional transmitter can interfere by detecting the conflict too late. Measurements on our outdoor testbed links and indoor testbed demonstrate a strong correlation between loss and volume of traffic from external sources on the same or adjacent channels [20].

This is different from the case of WiFi mesh networks like Roofnet [1], for which the authors concluded that multipath interference was a significant source of packet loss, while WiFi interference was not.

Other factors: Measurements on our testbed show that there is no measurable non-WiFi interference in our urban links [20]. This is indicated by the absence of significant correlation between noise floor (reported by the wireless card) and loss rates. Also, the loss rates on different channels are not correlated to each other implying the absence of any wide-band interfering noise. Experiments with different 802.11 PHY data rates showed that smaller data rates can have higher loss rates in many situations. This can be explained by the fact that packets at lower datarates take longer time on air and are thus more likely to collide with external traffic. Other studies by Raman *et al.* [7] show that weather conditions don't have noticeable effects on loss rates in long distance links.

2.5 Impact on TCP

Taken together, the protocol shortcomings of 802.11 and channel induced losses significantly lower end-to-end TCP performance. The use of stop-and-wait over long

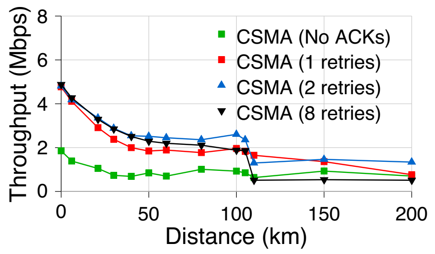


Figure 5: Cumulative throughput for TCP in both directions simultaneously over standard CSMA with 10% channel loss on emulated link. Traffic is 802.11b at PHY layer datarate of 11Mbps.

distances reduces channel utilization. In addition, we see correlated bursty collision losses due to interference from unsynchronized transmissions (over both single-link and multi-hop scenarios) as well as from external WiFi sources. Under these conditions, TCP flows often timeout resulting in very poor performance. The only configurable parameter in the driver is the number of packet retries. Setting a higher value on the number of retries decreases the loss rate, but at the cost of lower throughput resulting from lower channel utilization.

To better understand this trade-off, we measure the aggregate throughput of TCP flows in both directions on an emulated link while varying distance and introducing a channel packet loss rate of 10%. Figure 5 presents the aggregate TCP throughput with various number of MAC retries of the standard 802.11 MAC. Due to increased collisions and larger ACK turnaround times, throughput degrades gradually with increasing distances.

3 WiLDNet Design

In this section, we describe the design of WiLDNet and elaborate on how it addresses the 802.11 protocol shortcomings as well as achieves good performance in high-loss environments. In the previous section, we identified three basic problems with 802.11; (a) low utilization, (b) collisions at long distances, and (c) inter-link interference. To address the problem of low utilization, we propose the use of bulk packet acknowledgments (Section 3.1). To mitigate loss from collisions at long distances as well as inter-link interference, we replace the standard CSMA MAC with a TDMA-based MAC protocol. We build upon 2P [17] to adapt it to high-loss environments (Section 3.2). Additionally, to handle the challenge of high and variable packet losses, we design adaptive loss recovery mechanisms that use a combination of FEC and retransmissions with bulk acknowledgments (Section 3.3).

WiLDNet follows three main design principles. First, the system should not be narrowly focused to a single set of application types. It should be configurable to provide a broad tradeoff spectrum across different end-to-end properties including delay, bandwidth, loss, reliability and jitter. Second, all mechanisms proposed should be implementable on commodity off-the-shelf 802.11 cards. Third, the design should be lightweight, such that it can be implemented on the resource-constrained single-board computers (266-MHz CPU and 128 MB memory) used in our testbed.

3.1 Bulk Acknowledgments

We begin with the simple case of a single WiLD link, with each node having a half-duplex radio. As shown earlier, when propagation delays become longer, the default CSMA mechanism cannot determine whether the remote peer is sending a packet in time to back-off its own transmission and avoid collisions. Moreover, such a contention-based mechanism is overkill when precisely two hosts share the channel for a directional link.

Thus, a simple and efficient solution to avoid these collisions is to use an echo protocol between the sender and the receiver, which allows the two end-points to take turns sending and receiving packets. Hence, from a node's perspective, we divide time into send and receive time slots, with a burst of several packets being sent from one host to its peer in each slot.

Consequently, to improve link utilization, we replace the stock 802.11 stop-and-wait protocol with a sliding-window based flow-control approach in which we transmit a *bulk acknowledgment* (bulk ACK) from the receiver for a window of packets. We generate a bulk ACK as an aggregated acknowledgment for all the packets received within the previous slot. In this way, a sender can rapidly transmit a burst of packets rather than wait for an ACK after each packet.

The bulk ACK can be either piggybacked on data packets sent in the reverse direction, or sent as one or more stand-alone packets if no data packets are ready. Each bulk ACK contains the sequence number of the last packet received in order and a variable-length bit vector ACK for all packets following the in-order sequence. Here, the sequence number of a packet is locally defined between the pair of end-points of a WiLD link.

Like 802.11, the bulk ACK mechanism is not designed to guarantee perfect reliability. 802.11 has a maximum number of retries for every packet. Similarly, upon receiving a bulk ACK, the sender can choose to advance the sliding window skipping unacknowledged packets if the retry limit is exceeded. In practice, we support different retry limits for packets of different flows. The bulk ACK mechanism introduces packet reordering at the link layer, which may not be acceptable for TCP traffic. To handle this, we provide in-order packet delivery at the link layer either for the entire link or at a per-flow basis.

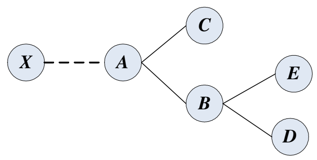


Figure 6: Example topology to compare synchronization of 2P and WiLDNet.

3.2 Designing TDMA on Lossy Channels

To address the inappropriateness of CSMA for WiLD networks, 2P [17] proposes a contention-free TDMA based channel access mechanism. 2P eliminates inter-link interference by synchronizing all the packet transmissions at a given node (along all links which operate on the same channel channel). In 2P, a node in transmission mode simultaneously transmits on all its links for a globally known specific period, and then explicitly notifies the end of its transmission period to each of its neighbors using marker packets. A receiving node waits for the marker packets from all its neighbors before switching over to transmission mode. In the event of a loss of a marker packet, a receiving node uses a timeout to switch into the transmission mode.

The design of 2P, while functional, is not well suited for lossy environments. Consider the simple example illustrated in Figure 6, where all links operate on the same channel. Consider the case where (X, A) is the link experiencing high packet loss-rate. Let T denote the value of the time-slot. Whenever a marker packet transmitted by X is lost, A begins transmission only after a timeout period T_0 ($\geq T$). This, in turn, delays the next set of transmissions from nodes B and C to their other neighbors by a time period that equals $T_0 - T$. Unfortunately, this propagation of delay does not end here. In the time slot that follows, D's transmission to its neighbors is delayed by $T_0 - T$. Hence, what we observe is that the loss of marker packets has a "ripple effect" in the entire network creating an idle period of $T_0 - T$ along every link. When markers along different links are dropped, the ripples from multiple links can interact with each other and cause more complex behavior.

Ideally, one would want $T_0 - T$ to be very small. If all nodes are perfectly time synchronized, we can set $T_0 = T$. However, in the absence of global time synchronization, one needs to set a conservative value for T_0. 2P chooses $T_0 = 1.25 \times T$. The loss of a marker packet leads to an idle period of $0.25 \times T$ (in 2P, this is 5 ms for $T = 20$ ms). In bursty losses, transmitting multiple marker packets may not suffice.

Given that many of the links in our network experience sustained loss-rates over 5–40%, in WiLDNet, we use an implicit synchronization approach that aims to reduce the value of $T_0 - T$. In WiLDNet, we use a simple loose time synchronization mechanism similar to the basic linear time synchronization protocol NTP [13], where during each time slot along each link, the sender acts as the master and the receiver as the slave. Consider a link (A, B) where A is the sender and B is the receiver at a given time. Let t_{send_A} and t_{recv_B} denote the start times of the slot as maintained by A and B. All the packets sent by A are timestamped with the time difference (δ) between the time the packet has been sent (t_1) and the beginning of the send slot(t_{send_A}). When a packet is received by B at time t_2, the beginning of B's receiving slot is adjusted accordingly: $t_{recv\_B} = t_2 - \delta$. As soon as B's receive slot is over, and $t_{send\_B} = t_{recv\_B} + T$ is reached, B starts sending for a period T.

Due to the propagation delay between A and B, the send and corresponding receive slots are slightly skewed. The end-effect of this loose synchronization is that the value of $T_0 - T$ is limited by the propagation delay across the link even with packet losses (assuming clock speeds are roughly comparable). Hence, an implicit synchronization approach significantly reduces the value of $T_0 - T$ thereby reducing the overall number of idle periods in the network.

3.3 Adaptive Loss Recovery

To achieve predictable end-to-end performance, it is essential to have a loss recovery mechanism that can hide the loss variability in the underlying channel. Achieving such an upper bound (q) on the loss-rate perceived by higher level applications is not easy in our settings. First, it is hard to predict the arrival and duration of bursts. Second, the loss distribution that we observed on our links is non-stationary even on long time scales (hourly and daily basis). Hence, a simple model cannot capture the channel loss characteristics.

In WiLDNet, we can either use retransmissions or FEC to deal with losses (or a combination of both). A retransmission based approach can achieve the loss-bound q with minimal throughput overhead but at the expense of increased delay. An FEC based approach incurs additional throughput overhead but does not incur a delay penalty especially since it is used in combination with TDMA on a per-slot basis. However, an FEC approach cannot achieve arbitrarily low loss-bounds mainly due to the unpredictability of the channel.

3.3.1 Tuning the Number of Retransmissions

To achieve a loss bound q independent of underlying channel loss rate $p(t)$, we need to tune the number of retransmissions. One can adjust the number of retransmissions $n(t)$ for a channel loss-rate $p(t)$ such that $(1 - p(t))^{n(t)} = q$. Given that our WiLD links support in-order delivery (on a per-flow or on whole link basis), a larger $n(t)$ also means a larger maximum delay, equal to $n(t) * T$ for a slot period T. One can set different values of $n(t)$ for different flows. We found that estimating $p(t)$ using an exponentially weighted average is sufficient in

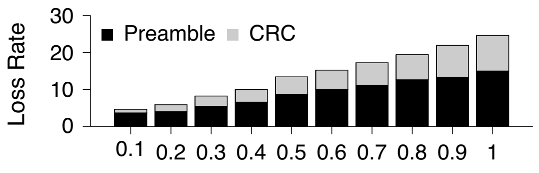


Figure 7: Proportion of CRC and preamble errors in channel loss. Traffic is at UDP CBR packets of 1440 bytes each at 802.11b PHY datarate of 11Mbps. Main link is sending at 2Mbps. The sending rate of the interferer increases from 0.1Mbps to 1Mbps.

our links to achieve the target loss estimate q. A purely retransmission based recovery mechanism has minimal throughput overhead as only the lost packets are retransmitted but this comes at a cost of high delay due to the long round-trip times over WiLD links.

3.3.2 Adaptive FEC-Based Recovery

Designing a good FEC mechanism in highly variable lossy conditions requires accurate estimation of the underlying channel loss. When the loss is underestimated, the redundant packets cannot be decoded at all making them useless, but overestimating the loss rate leads to unnecessary overhead.

Motivating inter-packet FEC: We can perform two types of FEC: inter-packet FEC (coding across packets) or intra-packet FEC (coding redundant blocks within a packet). Based on extensive measurements on a wireless channel emulator we observe that in presence of external WiFi interference, lost packets can be categorized into either CRC errors or preamble errors. A CRC error packet is received by the driver with a check sum error. However, an error in the preamble leads to the entire packet being dropped completely. Figure 7 shows the breakup of the loss rate with increasing external interference. We observe although the proportion of preamble errors decreases as external interference increases, it still causes at least 50% of all errors. Moreover a substantial number of the CRC error packets were truncated. We choose not to perform intra-packet FEC because it can only help recover packets that have CRC errors. Hence, we chose to perform inter-packet FEC.

Estimating redundancy: We apply FEC in combination with TDMA. For every time slot of N packets, we add $N-K$ redundant packets to K original packets. To estimate the redundancy factor, $r = (N-K)/K$, we choose a simple but not perfect estimation policy based on a weighted average of the losses observed in the previous M time slots. Here, we specifically chose a small value of $M = 10$ because it is hard to predict the start of a burst. Secondly, a small value of M, can quickly adapt to both the start and end of a loss burst saving unnecessary redundant FEC packets. For a time slot of $T = 10ms$, $M = 10$ corresponds to $200ms$ (with symmetric slot allocation in both directions) to adapt to a change in the loss behavior. Also due to non-stationary loss distributions, the benefit of using more complicated distribution based estimation approaches [22] is marginal. This type of FEC is best suited for handling residual losses and bursts that are longer than the time required for loss estimation mechanism to adapt.

4 Implementation

In this section, we describe the implementation details of WiLDNet. Our implementation comprises two parts: (a) driver-level modifications to control or disable features implemented in hardware (Section 4.1); (b) a *shim* layer that sits above the 802.11 MAC (Section 4.2) and uses the Click [11] modular router software to implement the functionalities described in Section 3.

4.1 Driver Modifications

The wireless cards we use in our implementation are the high power (200-400 mW) Atheros-based chipsets. To implement WiLDNet, we have to disable the following 802.11 MAC mechanisms:

•We disable **link-layer association** in Atheros chipsets using the *AdHoc-demo* mode.

•We disable **link layer retransmissions and automatic ACKs** by using 802.11 QoS frames with WMM extensions set to the no-ACK policy.

•We disable **CSMA** by turning off the Clear Channel Assessment (CCA) in Atheros chipsets. With CCA turned off, the radio card can transmit packets right away without waiting for a clear channel.

4.2 Software Architecture Modifications

In order to implement single-link and inter-link synchronization using TDMA, the various loss recovery mechanisms, sliding window flow control, and packet reordering for in-order delivery, we use the Click modular router [11] framework. We use Click because it enables us to prototype quickly a modular MAC layer by composing different Click elements together. It is also reasonably efficient for packet processing especially if loaded as a kernel module. Using kernel taps, Click creates fake network interfaces, such as *fake0* in Figure 8 and the kernel communicates with these virtual interfaces. Click allows us to intercept packets sent to this virtual interface and modify them before sending them on the real wireless interface and vice versa.

Figure 8 presents the structure of the Click elements of our layered system, with different functionality (and corresponding packet header processing) at various layers:

Incoming/Outgoing Queues: The mechanisms for sliding window packet flow, bulk ACKs, selective retransmission and reordering for in-order delivery are implemented by the incoming/outgoing queue pair. Packet buffering at the sender is necessary for retransmissions,

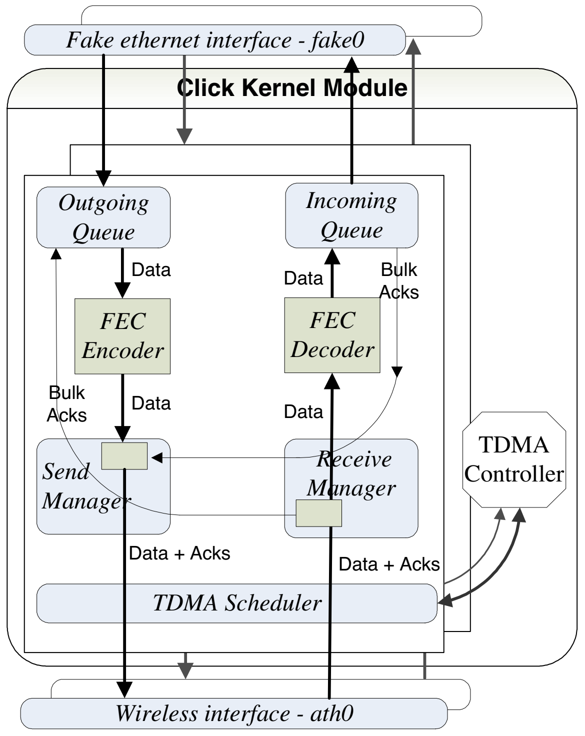


Figure 8: Click Module Data Flow

and buffering at the receiver enables reordering. In-order delivery and packet retransmission are optional, and the number of retries can be set on a per-packet basis.

FEC Encoder/Decoder: An optional layer is responsible for inter-packet forward error correction encoding and decoding. For our implementation we modify a FEC library [19] that uses erasure codes based on Vandermonde matrices computed over $GF(2^m)$. This FEC method uses a (K, N) scheme, where the first K packets are sent in their original form, and $N - K$ redundant packets are generated, for a total of N packets sent. At the receiver, the reception of any K out of the N packets enables the recovery of the original packets. We choose this scheme because, in loss-less situations, it introduces very low latency: the original K packets can be immediately sent by the encoder (without undergoing encoding), and immediately delivered to the application by the decoder (without undergoing decoding).

TDMA Scheduler and Controller: The Scheduler ensures that packets are being sent only during the designated send slots, and manages packet timestamps as part of the synchronization mechanism. The Controller implements synchronization among the wireless radios, by enforcing synchronous transmit and receive operation (all the radios on the same channel have a common send slot, followed by a common receive slot).

4.2.1 Timing issues

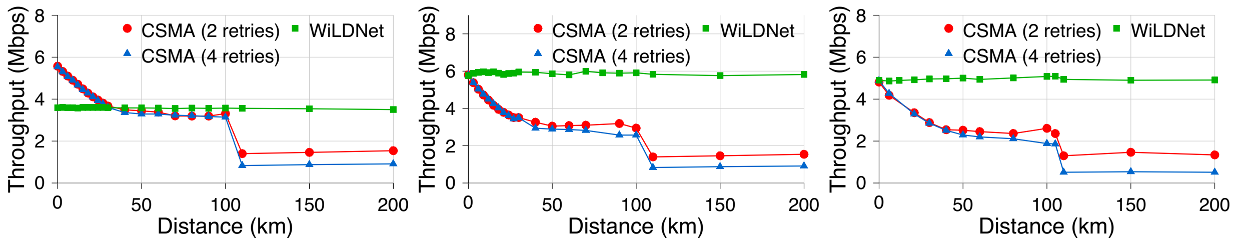
We do not use Click timers to implement time synchronization because the underlying kernel timers are not precise at the granularity of our time slots (10ms-40ms) on our hardware platform (266MHz CPU). Also packet queuing in the wireless interface causes variability in the time between the moment Click emits a packet and the time the packet is actually sent on the air interface. Thus, the propagation delay between the sending and the receiving click modules on the two hosts is not constant, affecting time slot calculations. Fortunately, this propagation delay is predictable for the first packet in the send slot, when the hardware interface queues are empty. Thus, in our current implementation, we only timestamp the first packet in a slot, and use it for adjusting the receive slot at the peer. If this packet is lost, the receiver's slot is not adjusted in the current slot, but since the drift is slow this does not have a significant impact. In the future we intend to perform this timestamping in the firmware - that would allow us to accurately timestamp every packet just before packet transmission.

Another timing complication is related to estimating whether we have time to send a new packet in the current send slot. Since the packets are queued in the wireless interface, the time when the packet leaves Click cannot be used to estimate this. To overcome this aspect, we use the notion of *virtual time*. At the beginning of a send slot, the virtual time t_v is same as current (system) time t_c. Every time we send a packet, we estimate the transmission time of the packet on the channel and recompute the virtual time: $t_v = max(t_c, t_v) + duration(packet)$. A packet is sent only after checking that the virtual time after sending this packet will not exceed the end of the send slot. Otherwise, we postpone the packet until the next slot.

5 Experimental Evaluation

The main goals of WiLDNet are to increase link utilization and to eliminate the various sources of packet loss observed in a typical multi-hop WiLD deployment, while simultaneously providing flexibility to meet different end-to-end application requirements. We believe these are the first actual implementation results over an outdoor multi-hop WiLD network deployment.

Raman *et al.* [17] show the improvements gained by the 2P-MAC protocol in simulation and in an indoor environment. However, a multi-hop outdoor deployment also has to deal with high losses from external interference. 2P in its current form does not have any built-in recovery mechanism and it is not clear how any recovery mechanism can be combined with the marker-based synchronization protocol. Hence, we do not have any direct comparison results with 2P on our outdoor wireless links. Also, the proof-of-concept implementation of 2P was for the Prism 2.5 wireless chipset and it would be non-trivial to implement the same in WiLDNet using features of the Atheros chipset.



(a) TCP flow in one direction (b) TCP flow in both directions (c) TCP in both directions, 10% channel loss

Figure 9: TCP throughput for WiLDNet vs 802.11 CSMA. Each measurement is for a TCP flow of 60s, 802.11b PHY, 11Mbps.

Our evaluation has three main parts:

- We analyze the ability of WiLDNet to maintain high performance (high link utilization) over long-distance WiLD links. At long distances, we demonstrate 2–5x improvements in cumulative throughput for TCP flows in both directions simultaneously.
- Next, we evaluate the ability of WiLDNet to scale to multiple hops and eliminate inter-link interference. WiLDNet yields a 2.5x improvement in TCP throughput on our real-world multi-hop setup.
- Finally, we evaluate the effectiveness of the two link recovery mechanisms of WiLDNet: Bulk Acks and FEC.

5.1 Single Link

In this section we demonstrate the ability of WiLDNet to eliminate link under-utilization and packet collisions over a single WiLD link. We compare the performance of WiLDNet (slot size of 20ms) with the standard 802.11 CSMA (2 retries) base case.

The first set of results show the improvement of WiLDNet on a single emulator link with increasing distance. Figure 9(a) compares the performance of TCP flowing only in one direction. The lower throughput of WiLDNet, approximately 50% of channel capacity, is due to symmetric slot allocation between the two end points of the link. However, over longer links (>50 km), the TDMA-based channel allocation avoids the under-utilization of the link as experienced by CSMA. Also, beyond 110 km (the maximum possible ACK timeout), the throughput with CSMA drops rapidly because of unnecessary retransmits (Section 2.3.1). Figure 9(b) shows the cumulative throughput of TCP flowing simultaneously in both directions. In this case, WiLDNet effectively eliminates all collisions occurring in presence of bidirectional traffic. TCP throughput of 6 Mbps is maintained for all distances.

Table 1 compares WiLDNet and CSMA for some of our outdoor wireless links. We show TCP throughput in one direction and the cumulative throughput for TCP simultaneously flowing in both directions. Since these are outdoor measurements, there is significant variation over time and we show both the mean and standard deviation for the measurements. We can see that as the link distance increases, the improvement of WiLDNet is more substantial. Infact, for the 65 km link in Ghana, WiLDNet's throughput at 5.5 Mbps is about 8x better than standard CSMA.

Link	Distance	Loss rates	802.11 CSMA (Mbps)		WiLDNet (Mbps)	
	(km)	(%)	One dir	Both dir	One dir	Both dir
B-R	8	3.4	5.03 (0.02)	**4.95** (0.03)	3.65 (0.01)	**5.86** (0.05)
P-S	45	2.6	3.62 (0.20)	**3.52** (0.17)	3.10 (0.05)	**4.91** (0.05)
Ghana	65	1.0	2.80 (0.20)	**0.68** (0.39)	2.98 (0.19)	**5.51** (0.07)

Table 1: Mean TCP throughput (flow in one direction and cumulative for both directions simultaneously) for WiLDNet and CSMA for various outdoor links (distance and loss rates). The standard deviation is shown in parenthesis for 10 measurements. Each measurement is for TCP flow of 30s at a 802.11b PHY-layer datarate of 11Mbps.

5.2 Multiple Hops

This section validates that WiLDNet eliminates inter-link interference by synchronizing receive and transmit slots in TDMA resulting in up to 2x TCP throughput improvements over standard 802.11 CSMA in multi-hop settings.

The first set of measurements were performed on our indoor setup (Section 2.2) where we recreated the conditions of a linear outdoor multi-hop topology using the RF isolation boxes. Thus transmissions from local radios interfere with each other but multiple local radio interfaces can receive simultaneously. We then measure TCP throughput of flows in the one direction and then both directions simultaneously for both standard 802.11 CSMA and WiLDNet (with slot size of 20ms). All the links were operating on the same channel. As we see in Table 2, as the number of hops increases, standard 802.11's TCP throughput drops substantially when transmissions from a radio collide with packet reception on a nearby local radio on the same node. WiLDNet avoids these collisions and maintains a much higher cumulative TCP throughput (up to 2x for the 3-hop setup) by proper synchronization of send and receive slots.

Linear setup	802.11 CSMA (Mbps)			WiLDNet (Mbps)		
	Dir 1	Dir 2	Both	Dir 1	Dir 2	Both
2 nodes	5.74 (0.01)	5.74 (0.01)	**6.00** (0.01)	3.56 (0.03)	3.53 (0.02)	**5.85** (0.07)
3 nodes	2.60 (0.01)	2.48 (0.01)	**2.62** (0.01)	3.12 (0.01)	3.12 (0.01)	**5.12** (0.03)
4 nodes	2.23 (0.01)	2.10 (0.01)	**1.99** (0.02)	2.95 (0.05)	2.98 (0.04)	**4.64** (0.24)

Table 2: Mean TCP throughput (flow in each direction and cumulative for both directions simultaneously) for WiLDNet and standard 802.11 CSMA. Measurements are for linear 2,3 and 4 node indoor setups recreating outdoor links running on the same channel. The standard deviation is shown in parenthesis for 10 measurements of flow of 60s each at 802.11b PHY layer datarate of 11Mbps.

Description (Mbps)	One direction	Both directions
Standard TCP: same channel	**2.17**	**2.11**
Standard TCP: diff channels	3.95	4.50
WiLD TCP: same channel	**3.12**	**4.86**
WiLD TCP: diff channels	3.14	4.90

Table 3: Mean TCP throughput (flow in single direction and cumulative for both directions simultaneously) comparison for WiLDNet and standard 802.11 CSMA over a 3-hop outdoor setup ($K \leftrightarrow P \leftrightarrow M$). Averaged over 10 measurements of TCP flow for 60s at 802.11b PHY layer datarate of 11Mbps.

We can also see that although WiLDNet has more than 2x improvement over standard 802.11, the final throughput (4.6Mbps) is still much smaller than the raw throughput of the link (6-7Mbps). This can be attributed to the overhead of synchronization and packet processing in Click running on our low-power (266MHz) single board routers. A more efficient synchronization mechanism implemented in the firmware (rather than Click) would deliver much better improvement.

We also measure this improvement on our outdoor testbed between the nodes K and M relayed through node P. We again compare the TCP throughput for WiLDNet and standard 802.11 CSMA with links operating on the same channel. In order to quantify the effect of inter-link interference, we also perform the same experiments with the links operating on different, non-overlapping channels, in which case the inter-link interference is almost zero, as previously shown in Figure 3.

We can see that, for same channel operation, the cumulative TCP throughput in both directions with WiLDNet (4.86 Mbps) is more than twice the throughput observed over standard 802.11 (2.11 Mbps). The improvement is substantially lower for the unidirectional case (3.14 Mbps versus 2.17 Mbps), because the WiLD links are constrained to send in one direction only roughly half of the time.

Another key observation is that WiLDNet is capable of eliminating almost all inter-link interference. This is shown by the fact that the throughput achieved by WiLDNet is almost the same, whether the links operate on the same channel or on non-overlapping channels.

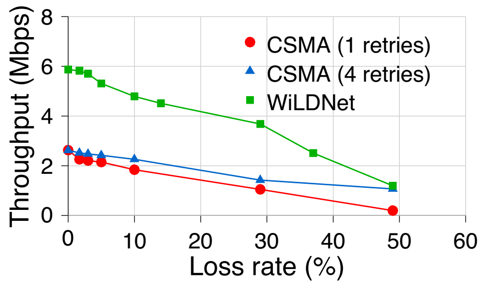


Figure 10: Comparison of cumulative throughput for TCP in both directions simultaneously for WiLDNet and standard 802.11 CSMA with increasing loss on 80km emulated link. Each measurement was for 60s TCP flows of 802.11b at 11Mbps PHY datarate.

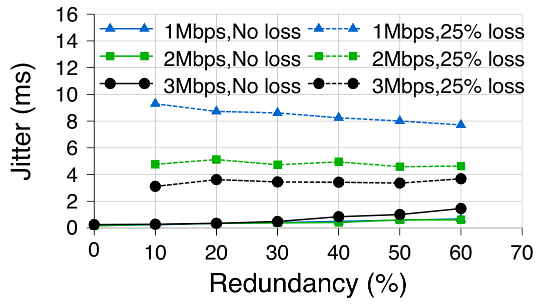


Figure 11: Jitter overhead of encoding and decoding for WiLDNet on single indoor link. Traffic is 1440 byte UDP CBR packets at PHY datarate of 11Mbps in 802.11b.

5.3 WiLDNet Link-Recovery Mechanisms

Our next set of experiments evaluate WiLDNet's adaptive link recovery mechanisms in conditions closer to the real world, where errors are generated by a combination of collisions and external interference. We evaluate both the bulk ACK and FEC recovery mechanisms.

5.3.1 Bulk ACK Recovery Mechanism

For our first experiment, presented in Figure 9(c), we vary the link length on the emulator, and we introduce a 10% error rate through external interference. We again measure the cumulative throughput of TCP flows in both directions for WiLDNet and standard 802.11 CSMA. As can be seen, WiLDNet maintains a constant throughput with increasing distance as opposed to the 802.11 CSMA. Due to the 10% error, WiLD incurs a constant throughput penalty of approximately 1 Mbps compared to the no-loss case in Figure 9(b).

In our second experiment we fix the distance in the emulator setup to 80 km, and vary channel loss rates. The results in Figure 10 show that WiLDNet maintains roughly a 2x improvement over standard CSMA's recovery mechanism for packet loss rates up to 30%.

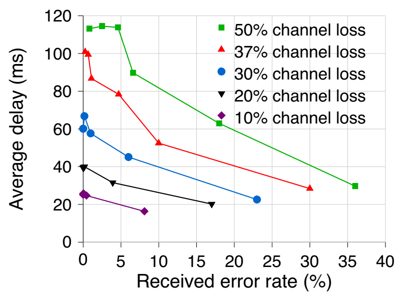


Figure 12: Avg. delay with decreasing target loss rate (X-axis) for various loss rates in WiLDNet on single emulated 60km link (slot size=20ms).

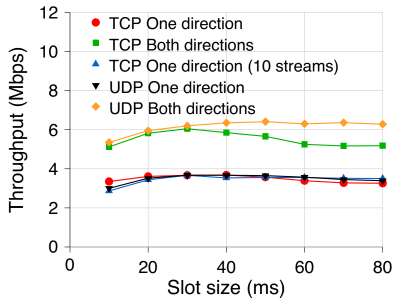


Figure 13: Throughput for increasing slot sizes (X-axis) in WiLDNet for various types of traffic on single emulated 60km link.

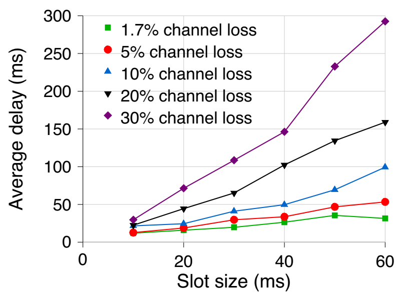


Figure 14: Avg. delay at increasing slot sizes (X-axis) for various loss rates in WiLDNet on single emulated 60km link.

5.3.2 Forward Error Correction (FEC)

To measure the jitter introduced by the FEC mechanism, we performed a simple experiment where we measured the jitter of a flow under two conditions: in the absence of any loss and in the presence of a 25% loss. Figure 11 illustrates the jitter introduced by WiLDNet's FEC implementation. We can see that in the absence of any loss, when only encoding occurs, the jitter is minimal. However, in the presence of loss, when decoding also takes place, the measured jitter increases. However, the magnitude of the jitter is very small and well within the acceptable limits of many interactive applications (voice or video), and decreases with higher throughputs (since the decoder waits less for redundant packets to arrive).

Moreover, considering the combination of FEC with TDMA, the delay overheads introduced by these methods overlap, since the slots when the host is not actively sending can be used to perform encoding without incurring any additional delay penalties.

6 Tradeoffs

One of the main design principles of WiLDNet is to build a system that can be configured to adapt to different application requirements. In this section we explore the tradeoff space of throughput, delay and delivered error rates by varying the slot size, number of bulk retransmissions and FEC redundancy parameters. We observe that WiLDNet can perform in a wide spectrum of the parameter space, and can easily be configured to meet specific application requirements.

6.1 Choosing number of retransmissions

The first tradeoff that we explore is choosing the number of retries to get a desired level of final error rate on a WiLD link. Although retransmission based loss recovery achieves optimal throughput utilization, it comes at a cost of increased delay; the loss rate can be reduced to zero by arbitrarily increasing the number of retransmissions at the cost of increased delay. This tradeoff is illustrated in Figure 12 which shows a plot of delay versus error rate for varying channel loss rates (10% to 50%). Retries are decreased from 10 to 0 from left to right for a given series in the figure. All the tests are with unidirectional UDP at 1 Mbps for a fixed slot size of 20ms on a single emulator 60km link. We can see that as we try to reduce the final error rate at the receiver, we have to use more retries and this increases the average delay. In addition, we also observe that larger the number of retries, larger the end-to-end jitter (especially at higher loss rates).

This tradeoff has important implications for applications that are more sensitive to delay and jitter (such as real time audio and video) as compared to applications which require high reliability. For such applications, we can achieve a balance between the final error rate and the average delay by choosing an appropriate retry limit. For applications that require improved loss characteristics without incurring a delay penalty, we need to use FEC for loss recovery.

6.2 Choosing slot size

The second tradeoff that we explore is the effect of slot size on TCP and UDP throughput . Our experiments are performed on a 60-km emulated link (Figure 13). As discussed in Section 3.2, switching between send and receive slots incurs a non-negligible overhead for the Click based WiLDNet implementation. This overhead although constant for all slot sizes, occupies a higher fraction of the slot for smaller slots sizes. As a consequence, at small slot sizes the achieved throughput is lower. However, the UDP throughput levels off beyond a slot size of 20 ms. We also observe the TCP throughput reducing slightly at higher slot size. This is because the bandwidth-delay product of the link increases with slot size, but the send TCP window sizes are fixed. UDP throughput does not decrease at higher slot sizes.

In the next experiment, we measure the average UDP packet transmission delay while varying the slot size, for several channel error rates. The results are presented in Figure 14; each series represents a unidirectional UDP test (1 Mbps CBR) at a particular channel loss rate with

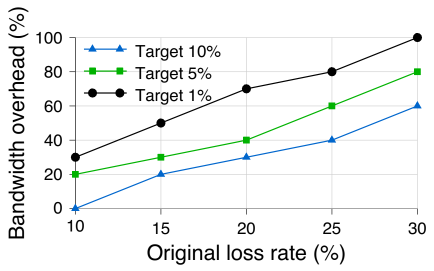


Figure 15: Throughput overhead vs channel loss rate for FEC on single emulated 20km link. Traffic is 1Mbps CBR UDP.

WiLDNet using maximum number of retries. Figure 14 shows the increase in delay with increasing slot size. It is clear that slot sizes beyond 20 ms do not result in substantially higher throughputs, but they do result in much larger delay. However, if lower delay is required, smaller slots can be used at the expense of some throughput overhead consumed by the switching between the transmit and receive modes.

6.3 Choosing FEC parameters

The primary tunable FEC parameter is the redundancy factor $r = (N - K)/K$, also referred to as throughput overhead. Although FEC incurs a higher throughput overhead than retransmissions, it incurs a smaller delay penalty as illustrated earlier in Section 5.3.2. To analyze the tradeoff between FEC throughput overhead and the target loss-rate, we consider the case of a single WiLD link (in our emulator environment) with a simple Bernoulli loss-model (every packet is dropped with probability p). Figure 15 shows the amount of redundancy required to meet three different target loss-rates of 10%, 5% or 1% as the raw channel error rates (namely p) increase. We see that in order to achieve very low target loss-rates, a lot of redundancy is required (for example, FEC incurs a 100% overhead to reduce the loss-rate from 30% to 1%). Also, when a channel is very bursty and has an unpredictable burst arrival pattern, it is very hard for FEC to achieve arbitrarily low target loss-rates.

For applications that can tolerate one round of retransmissions, we can use a combination of FEC and retransmissions to provide a tradeoff between overall throughput overhead, delay and target loss-rate. In the case of a channel with a stationary loss distribution, OverQoS [22] shows that the optimal policy to minimize overhead is to not use FEC in the first round but use it in the second round to pad retransmission packets. With unpredictable and highly varying channel loss conditions, an alternative promising strategy is to use FEC in the first round during bursty periods to reduce the perceived loss-rate.

7 Related Work

Long Distance WiFi: The use of 802.11 for long distance networking with directional links and multiple radios per node, raises a new set of technical issues that were first illustrated in [4]. Raman *et al.*built upon this work in [17, 16] and proposed the 2P MAC protocol. WiLDNet builds upon 2P to make it robust in high loss environments. Specifically we modify 2P's implicit synchronization mechanism as well as build in adaptive bulk ACK based and FEC based link recovery mechanisms.

Other wireless loss recovery mechanisms: There is a large body of research literature in wireless and wireline networks that have studied the tradeoffs between different forms of loss recovery mechanisms. Many of the classic error control mechanisms are summarized in the book by Lin and Costello [12]. OverQoS [22] performs recovery by analyzing the FEC/ARQ tradeoff in variable channel conditions and the Vandermonde codes are used for reliable multicast in wireless environments [19].

Of particular interest for this work are the Berkeley Snoop protocol [2] which provides transport-aware link-layer recovery mechanisms in wireless environments. To compare the WiLDNet bulk ACK recovery mechanism with recovery at a higher layer, we experimented with a version of the original Snoop protocol [3] that we modified to run on WiLD links. Basically, each WiLD router ran one half of Snoop, the fixed host to mobile host part, for each each outgoing link and integrated all the Snoops on different links into one module.

We measured the performance of modified Snoop as a recovery mechanism over both standard 802.11 (CSMA) and over WiLDNet with no retries. We found that WiLDNet was still 2x better than Snoop. We also saw that Snoop was better than vanilla CSMA only at lower error rates (less than 10%). Thus, this indicates that higher layer recovery mechanisms might be better than stock 802.11 protocol, but only at lower error rates.

Other WiFi-based MAC protocols: Several recent efforts have focused on leveraging off-the-shelf 802.11 hardware to design new MAC protocols. Overlay MAC Layer (OML) [18] provides a deployable approach towards implementing a TDMA style MAC on top of the 802.11 MAC using loosely-synchronized clocks to provide applications and competing nodes better control over the allocation of time-slots. SoftMAC [14] is another platform to build experimental MAC protocols. MultiMAC [10] builds on SoftMac to provide a platform where multiple MAC layers co-exist in the network stack and any one can be chosen on a per-packet basis.

WiMax: An alternative to WiLD networks is WiMax [25]. WiMax does present many strengths over a WiFi: configurable channel spectrum width, better modulation (especially for non-line of sight scenarios), operation in licensed spectrum with higher transmit power, and thus longer distances. On the other hand, WiMax currently is primarily intended for carriers (like cellular) and does not support point-to-point operation. In addition, WiMax base-stations are expensive ($10,000) and the high spectrum license costs in most countries dissuades grassroots style deployments. Currently it is also very difficult to obtain licenses for experimental deployment and we are not aware of open-source drivers for WiMax base-stations and clients. However, most of our work in loss recovery and adaptive FEC would be equally valid for any PHY layer (WiFi or WiMax). With appropriate modifications and cost reductions, WiMax can serve as a more suitable PHY layer for WiLD networks.

8 Future Work and Conclusion

The commoditization of WiFi (802.11 MAC) hardware has made WiLD networks an extremely cost-effective option for providing network connectivity, especially in rural regions in developing countries. However providing coverage at high performance in real-world WiLD network deployments raises many research challenges: optimal planning and placement of long distance links, design of appropriate MAC and network protocols to provide quality of service to a wide variety of applications, remote management and fault tolerance to handle unpredictable node and link failures [23].

One of the most important challenges in this space is the sub-optimal performance of the standard 802.11 MAC protocol. In this paper, we identify the set of link- and MAC-layer modifications essential for achieving high throughput in multi-hop WiLD networks. Specifically, using a detailed performance evaluation, we show that the conventional 802.11 protocol is ill-suited for WiLD settings. Our proposed solution provides a 2-5x improvement in TCP throughput over the conventional 802.11 MAC.

Although this constitutes a substantial improvement, designing decentralized TDMA slot scheduling schemes for multi-hop and multi-channel networks to achieve optimal bandwidth and delay characteristics for realistic real-world asymmetric traffic demands is a significant future research direction. Our current solution builds the basic link mechanisms to provide quality of service. We intend to build end-to-end QoS solutions that leverage these mechanisms and adapt to a realistic traffic mix.

Encouraged by our initial results on our long distance outdoor testbed, we will now implement these modifications in our live rural deployments in India and Ghana. We expect that these improvements can have significant impact in accelerating the penetration of feasible network connectivity options in developing regions.

Acknowledgments

We thank the whole TIER group at UC Berkeley and Intel Research, Berkeley for braving wind, sun, rain and crazy taxi drivers to set up long-distance links in India, Ghana and the Bay Area so that we can run our experiments. We thank all the anonymous reviewers for their feedback and owe our gratitude to Brad Karp, our NSDI shepherd for his valuable comments which helped in substantially improving the paper.

References

[1] D. Aguayo, J. Bicket, S. Biswas, G. Judd, and R. Morris. Link-level Measurements from an 802.11b Mesh Network. In *ACM SIGCOMM*, Aug. 2004.

[2] H. Balakrishan. *Challenges to Reliable Data Transport over Heterogeneous Wireless Networks*. PhD thesis, University of California at Berkeley, Aug. 1998.

[3] H. Balakrishnan, S. Seshan, E. Amir, and R. Katz. Improving TCP/IP Performance over Wireless Networks. In *ACM MOBICOM*, Nov. 1995.

[4] P. Bhagwat, B. Raman, and D. Sanghi. Turning 802.11 Inside-out. *ACM SIGCOMM CCR*, 2004.

[5] S. Biswas and R. Morris. Opportunistic Routing in Multi-Hop Wireless Networks. *Hotnets-II*, November 2003.

[6] E. Brewer. Technology Insights for Rural Connectivity. Oct. 2005.

[7] K. Chebrolu, B. Raman, and S. Sen. Long-Distance 802.11b Links: Performance Measurements and Experience. In *ACM MOBICOM*, 2006.

[8] Connecting Rural Communities with WiFi. http://www.crc.net.nz.

[9] Digital Gangetic Plains. http://www.iitk.ac.in/mladgp/.

[10] C. Doerr, M. Neufeld, J. Filfield, T. Weingart, D. C. Sicker, and D. Grunwald. MultiMAC - An Adaptive MAC Framework for Dynamic Radio Networking. In *IEEE DySPAN*, Nov. 2005.

[11] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[12] S. Lin and D. Costello. *Error Control Coding: Fundamentals and Applications*. Prentice Hall, 1983.

[13] D. L. Mills. Internet Time Synchronization: The Network Time Protocol. In *Global States and Time in Distributed Systems, IEEE Computer Society Press*. 1994.

[14] M. Neufeld, J. Fifield, C. Doerr, A. Sheth, and D. Grunwald. SoftMAC - Flexible Wireless Research Platform. In *HotNets-IV*, Nov. 2005.

[15] Partnership for Higher Education in Africa. Securing the Linchpin: More Bandwidth at Lower Cost, 2006.

[16] B. Raman and K. Chebrolu. Revisiting MAC Design for an 802.11-based Mesh Network. In *HotNets-III*, 2004.

[17] B. Raman and K. Chebrolu. Design and Evaluation of a new MAC Protocol for Long-Distance 802.11 Mesh Networks. In *ACM MOBICOM*, Aug. 2005.

[18] A. Rao and I. Stoica. An Overlay MAC layer for 802.11 Networks. In *MOBISYS*, Seattle,WA, USA, June 2005.

[19] L. Rizzo. Effective Erasure codes for Reliable Computer Communication Protocols. *ACM CCR*, 1997.

[20] A. Sheth, S. Nedevschi, R. Patra, S. Surana, L. Subramanian, and E. Brewer. Packet Loss Characterization in WiFi-based Long Distance Networks. *IEEE INFOCOM*, 2007.

[21] Spirent Communications. http://www.spirentcom.com.

[22] L. Subramanian, I. Stoica, H. Balakrishnan, and R. Katz. OverQoS: An Overlay Based Architecture for Enhancing Internet QoS. In *USENIX/ACM NSDI*, March 2004.

[23] L. Subramanian, S. Surana, R. Patra, M. Ho, A. Sheth, and E. Brewer. Rethinking Wireless for the Developing World. *Hotnets-V*, 2006.

[24] The Akshaya E-Literacy Project. http://www.akshaya.net.

[25] WiMAX forum. http://www.wimaxforum.org.

S4: Small State and Small Stretch Routing Protocol for Large Wireless Sensor Networks

Yun Mao$^+$ Feng Wang* Lili Qiu* Simon S. Lam* Jonathan M. Smith$^+$

The University of Texas at Austin* University of Pennsylvania$^+$

Abstract

Routing protocols for wireless sensor networks must address the challenges of reliable packet delivery at increasingly large scale and highly constrained node resources. Attempts to limit node state can result in undesirable worst-case routing performance, as measured by stretch, which is the ratio of the hop count of the selected path to that of the optimal path.

We present a new routing protocol, Small State and Small Stretch (S4),which jointly minimizes the state and stretch. S4 uses a combination of beacon distance-vector based global routing state and scoped distance-vector based local routing state to achieve a worst-case stretch of 3 using $O(\sqrt{N})$ routing state per node in an N-node network. Its average routing stretch is close to 1. S4 further incorporates local failure recovery to achieve resilience to dynamic topology changes. We use multiple simulation environments to assess performance claims at scale, and use experiments in a 42-node wireless sensor network testbed to evaluate performance under realistic RF and failure dynamics. The results show that S4 achieves scalability, efficiency, and resilience in a wide range of scenarios.

1 Introduction

Routing finds paths in a network along which to send data. It is one of the basic network functionalities. The effectiveness of routing protocols directly affects network scalability, efficiency, and reliability. With continuing growth of wireless network sizes, it is increasingly important to develop routing protocols that *simultaneously* achieve the following design goals.

- Small routing state: Using small amounts of routing state is essential to achieving network scalability. Many wireless devices are resource constrained. For example, mica2 sensor motes have only 4KB RAM. Limiting routing state is necessary for such devices to form large networks. Moreover, limiting routing state also helps to reduce control traffic used in route setup and maintenance, since the amount of routing state and control traffic is often correlated.
- Small routing stretch: Routing stretch is defined as the ratio between the cost of selected route and the cost of optimal route. Small routing stretch means that the selected route is efficient compared to the optimal route. It is a key quantitative measure of route *quality*, and affects global resource consumption, delay, and reliability.
- Resilience: Wireless networks often experience frequent topology changes arising from battery outage, node failures, and environmental changes. Routing protocols should find efficient routes even in the presence of such changes.

Existing routing protocols either achieve small worst-case routing stretches with large routing state (*e.g.*, shortest path routing) or achieve small routing state at the cost of large worst-case routing stretches (*e.g.*, geographic routing and hierarchical routing). In this paper, we present the design and implementation of Small State and Small Stretch (S4), a new addition to the routing protocol design space. S4 achieves a desirable balance among these characteristics, and is well suited to the wireless sensor network setting.

We make the following contributions.

1. S4 is the first routing protocol that achieves a worst-case routing stretch of 3 in large wireless networks. Its average routing stretch is close to 1.
2. S4's distance guided local failure recovery scheme significantly enhances network resilience, and is portable to other settings.
3. S4's scalability, effectiveness of resource use, and resilience are validated using multiple simulation environments and a 42-node sensor network testbed.

The rest of the paper is organized as follows. In Section 2, we discuss the limitations of previous work. We present the S4 routing protocol in Section 3. We evaluate its performance using high-level simulation in Section 4, to study the performance under ideal wireless environment with no wireless medium losses or collisions. In Section 5, we present evaluation results using TOSSIM, a packet-level simulator that models wireless medium and collisions, to study the performance in more realistic large-scale wireless networks. In Section 6 we describe testbed evaluation. We conclude in Section 7.

2 Related Work

Routing is a well-studied problem, but wireless sensor networks have introduced new challenges. Shortest path routing protocols (*e.g.*, DSR [10], AODV [22], DSDV [21]) can find good routes, but are limited in scale by both control traffic and the amount of state required at each node. Consequently, routing in large-scale wireless networks has focused on minimizing storage and exchange of routing state, and can be divided into geographic routing and hierarchical routing approaches.

In geographic routing, each node is assigned a coordinate reflecting its position in the network. Upon receiving a packet, a node selects a next hop closer to the destination in the coordinate space. Some geographic routing protocols use geographic locations as node coordinates, while others use virtual coordinates based on network proximity. As connectivity in the coordinate spaces is not complete, these schemes must address getting "stuck" in a local minimum, where no neighbor is closer to the destination than the current node. Some proposals such as GFG [1], GPSR [11], GOAFR+ [14], GPVFR [17] and variants use face traversal schemes that route packets on a planar graph derived from the original connectivity graph. Their delivery guarantees [5] depend on the assumption that the planarization algorithms (e.g. GG [6] and RNG [27]) can successfully planarize *any* network graph. These planarization algorithms typically assume a unit disk or quasi-unit disk model. However, these models can be inadequate for real wireless environments due to obstacles and multi-path fading. Kim, et. al [13] have shown that model failures in real radio environments can cause routing pathologies and persistent routing failures. CLDP [12] addresses the imperfect RF propagation problem using a right-hand probing rule to detect link-crossings and remove them to re-planarize the graph. GDSTR [16] provides delivery guarantee without requiring planarization by avoiding routing across the face of planar graphs and instead routing packets through a spanning tree.

The geographic coordinate-based routing schemes have at least three difficulties for wireless sensor networks. First, accurate geolocation either requires careful static setting or access to GPS, with consequences for cost and need for line-of-sight to satellites. Second, geographic distances may lack predictive value for network performance (*e.g.*, loss rate). This may result in paths with poor performance. Third, even with GPS and ideal radios, the best routing stretch for geographic routing is $O(c^2)$ in GOAFR+ [14] and ARF [15], where c is the length of the optimal path, and example topologies exist where this bound is tight [15].

Virtual coordinates reflecting underlying network connectivity address the first two difficulties, but still face the challenge of "dead ends", for which a recovery scheme is required. In addition, the overhead of computing and storing virtual coordinates is not negligible. For example, NoGeo [24] uses $O(\sqrt{N})$ *perimeter* nodes to flood the N-node network so that every node can learn its distances to all the perimeter nodes. Each node determines its virtual coordinate based on the distances to the perimeter nodes. However, perimeter nodes need to store $O(N)$ pair-wise distance amongst them, which is not scalable in large wireless networks with limited memory space per node. GEM [20] achieves greater scalability by using triangulation from a root node and two other reference nodes. However, the routing stretch is larger than that typical of geographic routing algorithms, and there is the additional cost of recomputing routing labels resulting from network failures. Fonseca, *et al.* [4] have proposed Beacon Vector Routing (BVR) which selects a few beacon nodes, and uses flooding to construct spanning trees from the beacons to all other nodes. A node's coordinate is a vector of distances from the node to all beacons, and each node maintains the coordinates of its neighbors. BVR defines a distance metric over these beacon vectors, and a node routes packets to the one that minimizes the distance. When greedy routing stalls, it forwards the packet towards the beacon closest to the destination. If the beacon still fails to make greedy progress, scoped flooding is used. None of the virtual coordinate-based routing algorithms provide worst-case routing stretch guarantees.

Hierarchical routing is an alternative approach to achieving scalability. Example protocols in this category include landmark routing [28], LANMAR [7], ZRP [8] and Safari [23]. Hierarchical routing protocols provide no guarantee on the routing stretch due to boundary effects: two nodes that are physically close may belong to different clusters or zones, and hence the route between them has to go through cluster heads, which can be arbitrarily longer than their shortest path.

Caesar et al. develop VRR [2], a scheme for layer-3 any-to-any routing based on distributed hash tables. To route to its successors on the virtual ring, a node sets up and maintains forwarding entries to its successors and predecessors along multi-hop physical paths. As a result, a node has both routing table entries towards its neighbors in the ring and also entries for the nodes on the paths in between. VRR greedily forwards a packet toward the node in the routing table with the closest ID to the destination ID. The routing state per node is roughly $O(\sqrt{N})$. Unlike S4, VRR does not provide worst-case routing stretch guarantee.

Theoretical work [3, 26] on achieving scalable and efficient routing has developed *compact routing* algorithms that provide a worst-case routing stretch of 3 while using at most $O(\sqrt{N \log N})$ state in an N-node network. This worst-case routing stretch is provably optimal when each node uses less than linear routing state [3, 26]. While compact routing seems to be a promising direction for large-scale networks, it cannot be directly translated into a routing protocol in a distributed network. In particular, the proposed algorithms do not specify how each node should build and maintain routing state for local clusters and for beacon nodes. Moreover, the algorithm in [26] requires choosing beacon nodes offline, considers only initial route construction, and cannot cope with topology changes, which precludes realization in our network setting. The implications of compact routing for average routing stretch also remain unclear.

3 S4 Routing Protocol

S4 uses the theoretical ideas of the compact routing algorithm [26] as a basis, refined by the addition of new techniques needed to obtain a practical routing protocol for large-scale wireless networks. We first describe the basic routing algorithm and note challenges for routing protocol design, and then present the S4 routing protocol. Throughout this paper, our metric for the cost of a route is the number of links traversed (*i.e.*, hop count).

3.1 Basic Routing Algorithm

In S4, a random set of nodes, L, are chosen as beacons. For a node d, let $L(d)$ denote the beacon closest to node d, and let $\delta(s,d)$ denote the shortest path distance from s to d. Each node s constructs the following local cluster, denoted as $C_k(s)$.

$$C_k(s) = \{c \in V | \delta(c,s) \leq k * \delta(c, L(c))\}, k \geq 1.$$

where V is the set of all nodes in the network. A local cluster of node s consists of all nodes whose distances to s are within k times their distances to their closest beacons. Each node s then maintains a routing table for all beacon nodes and nodes in its own cluster $C_k(s)$.

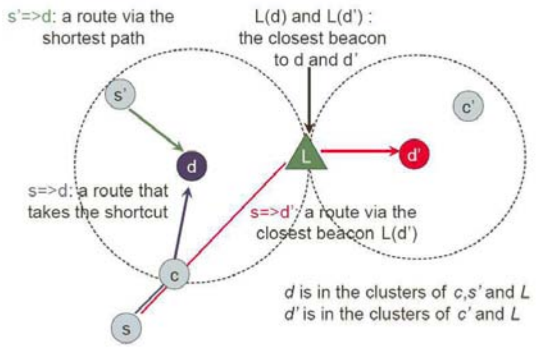

s'=>d: a route via the shortest path
L(d) and L(d') : the closest beacon to d and d'
s'
c'
d
L
d'
s=>d: a route that takes the shortcut
s=>d': a route via the closest beacon L(d')
c
s
d is in the clusters of c,s' and L
d' is in the clusters of c' and L

Figure 1: S4 routing examples. Every node within the circle of d has d in its local cluster. The route $s' \rightarrow d$ is the shortest path; the route $s \rightarrow d$ takes a shortcut at c before reaching $L(d)$; the route $s \rightarrow d'$ is through $L(d')$ without shortcut.

As shown in Figure 1, when routing from node s to node d, if $d \in C_k(s)$, we can directly use the shortest path to route from s to d. Otherwise, s first takes the shortest path towards $L(d)$, and then use the shortest path to route towards d. In the second case, the route does not have to always reach $L(d)$ before routing to d. Whenever data reaches a node c whose cluster contains d, c can directly route to d using the shortest path from c to d. According to the triangle inequality, the "shortcut" strictly improves routing stretch. We give the following theorem as an extension to the proof in [3, 26], in which a special case $k = 1$ is proved. Refer to our technical report [19] for the proof.

Theorem 1 *Let $C_k(s) = \{c \in V|\delta(c,s) < k * \delta(c, L(c))\}$, where $k \geq 1$. If each node s maintains next-hop for the shortest path to every beacon and every node in $C_k(s)$, the worst-case routing stretch is $1 + \frac{2}{k}$.*

As a special case, when $k = 1$, a local cluster of node s consists of all nodes whose distances to s are closer than their distances to their closest beacons. This special case is called compact routing [3, 26]. It is particularly interesting, since it has low worst-case storage cost of $O(\sqrt{N \log N})$ and provides a worst-case routing stretch of 3. In the remaining paper we consider $k = 1$, since it gives small routing state.

Practical concerns dictate three changes to the TZ compact routing scheme [26] to achieve S4. First, the boundary conditions of the cluster definitions are slightly different. In S4, $C(s) = \{c \in V|\delta(c,s) \leq \delta(c, L(c))\}$, but in the TZ scheme, $C(s) = \{c \in V|\delta(c,s) < \delta(c, L(c))\}$. That is, node c is in the cluster of s in S4 but not in the TZ scheme, if $\delta(c,s) = \delta(c, L(c))$. This change does not affect the worst-case routing stretch, and reduces average-case routing stretch at the cost of increasing routing state.

Second, to route towards node d, only $L(d)$ should be carried in the packet header as the location information in S4. In comparison, the TZ scheme requires a $label(d) = (L(d), port(L(d), d))$ for each packet, where $port(L(d), d)$ is the next hop at $L(d)$ towards d. Only with the label carried in the packet header, a beacon node can forward a packet towards d using next hop $port(L(d), d)$. It is necessary in the TZ scheme because the beacon nodes do not store routing state. However, in S4, as a result of the boundary condition change, each beacon node L stores routing state to all the nodes that have L as its closest beacon node. Given that the total storage cost of the additional field $port(L(d), d)$ in the labels is the same as the total number of routing entries at beacon nodes in S4 (*i.e.*, both are N), we favor storing routing state at beacon nodes since it reduces packet header length and the frequency of updating labels. The frequency of label updates is reduced because labels are updated only when $L(d)$ changes but not when $port(L(d), d)$ changes.

Finally, the TZ scheme proposes a centralized beacon node selection algorithm to meet expected worst case storage bound $O(\sqrt{NlogN})$ in an N-node network. Since practicality is our main design goal, in S4 we randomly select beacon nodes in a distributed fashion. It is proved that when $O(\sqrt{N})$ nodes are randomly selected as beacon nodes, the average storage cost on each node is still $O(\sqrt{N})$ [25]. As our evaluation results show, the storage cost is still low even for the worst cases. Note that the worst-case routing stretch of 3 still holds under random beacon node selection.

3.2 Design Challenges

Designing a routing protocol to realize the algorithm proposed in Section 3.1 poses the following challenges:

First, how to construct and maintain routing state for a local cluster? Frequent topology changes in wireless networks make it necessary to support incremental routing updates. Unlike traditional hierarchical routing, each node has its own cluster in compact routing. Therefore naive routing maintenance could incur significant overhead.

Second, how to construct and maintain routing state for beacon nodes? Knowledge of next-hops and shortest path distances to beacon nodes is important to the performance of S4. When beacon packets are lost, the routing state could be inaccurate, which could substantially degrade the performance.

Third, how to provide resilience against node/link failures and environmental changes? Maintaining up-to-date routing state could be expensive especially in a large network. Moreover routing changes take time to propagate. During the transient period (e.g., the period from the time when failure occurs to the time when the routing tables at all nodes are updated to account for the failure), many packets could be lost without a failure recovery scheme.

To address the above challenges, S4 consists of the following three major components: (i) scoped distance vector for building and maintaining routing state to nodes within a cluster, (ii) resilient beacon distance vector for efficient routing towards beacon nodes and facilitating inter-cluster routing, and (iii) distance guided local failure recovery for providing high quality routes even under dynamic topology changes. Below we will describe these three components in turn.

3.3 Intra-Cluster Routing: Scoped Distance Vector (SDV)

In S4, node s uses the shortest paths to route towards nodes in the cluster of s. Unlike the traditional hierarchical routing, in S4 each node s has its own cluster, which consists of nodes close to node s. This clustering is essential for providing a routing stretch guarantee, since it avoids boundary effects. In comparison, hierarchical routing cannot provide routing stretch guarantee due to boundary effects, where two nearby nodes belong to different clusters and the hierarchical route between them could be much longer than their direct shortest path.

A natural approach to building a local routing table is to use scoped flooding. That is, each node d floods the network up to $\delta(d, L(d))$ hops away from d, where $\delta(d, L(d))$ is the distance between d and its closest beacon $L(d)$. Scoped flooding works fine when the network is initialized, or when there are new nodes joining the network. But it is costly to send frequent scoped floodings to reflect constant topology changes, which often arises in wireless networks due to battery outage, node failures, and environmental changes.

Scoped distance vector: To provide cheap incremental routing updates, we propose using scoped distance vector (SDV) for constructing routing tables for local clusters. SDV is attractive because it is fully distributed, asynchronous, and supports incremental routing updates. SDV is more efficient than scoped flooding especially under small changes in a network topology, because a node in SDV propagates routing update only when its distance vector changes while in scoped flooding a node propagates a flooded packet regardless of whether its distance and next hop to a destination have changed.

In S4, each node s stores a distance vector for each destination d in its cluster as the following tuple:

$$< d, nexthop(s,d), \delta(s,d), seqno(d), scope(d), updated >$$

where d and $nexthop(s,d)$ are both node IDs, $seqno$ is the latest sequence number for destination d, and $scope(d)$ is the distance between d and d's closest beacon, and $updated$ is whether the distance vector has been updated since the last routing update.

A node s exchanges its distance vectors with its neighbors either synchronously or asynchronously. Node s initializes $\delta(s,c) = 1$ for only $c \in neighbor(s)$, and ∞ otherwise. Upon receiving a distance vector, a node c uses the newly received distance vectors to update its routing state. Node c further propagates the update for s only when its current distance from s is below $scope(s)$ and its distance vector to s has changed.

Benefits of SDV: SDV supports incremental routing updates. This allows a wireless network to dynamically adapt to routing changes. Moreover, unlike traditional distance vector protocols, SDV does not suffer from the count-to-infinity problem,[1] because the scope is typically small (*e.g.*, We evaluate a 1000-node network with 32 beacons, and its average scope is 3.35 and maximum scope is 13. This implies routing loops can be detected within 13 hops).

3.4 Inter-Cluster Routing: Resilient Beacon Distance Vector (RBDV)

To support routing across clusters, each node is required to know its distances to all beacons. This can be achieved

[1]The count-to-infinity problem is that when a link fails, it may take a long time (on the order of network diameter) before the protocol detects the failure. During the interim routing loops may exist.

by constructing a spanning tree rooted from each beacon nodes to every other node in the network. Flooding beacon packets reliably is important to the routing performance, because loss of beacon packets may introduce errors in estimating the closest beacon and its distance, and degrade the performance of S4. We develop a simple approach to enhance resilience of beacon packets.

Routing state construction and maintenance: To construct routing state for beacon nodes, every beacon periodically broadcasts beacon packets, which are flooded throughout the network. Every node then keeps track of the shortest hop count and next-hop towards each beacon.

Since beacon packets are broadcast and typical MAC protocols (*e.g.*, CC1000 used in sensor motes) do not provide reliability for broadcast packets, it is essential to enhance the resilience of beacon packets at the network layer. Our idea is to have a sender retransmit the broadcast packet P until T neighbors have forwarded P or until the maximum retry count $Retry_{max}$ is reached. T and $Retry_{max}$ provide a tradeoff between overhead and reliability. In our evaluation, we use $Retry_{max} = 3$, $T = 100\%$ for beacon nodes, and $T = 1/3$ for non-beacon nodes. $T = 100\%$ for a beacon node is used because all neighbors of the beacon nodes should forward the beacon packet. In comparison, for a non-beacon node c, only a subset of c's neighbors are farther away from the beacon than c and need to forward the beacon packet received from c. Therefore we use a smaller T for non-beacon nodes.

3.5 Distance Guided Local Failure Recovery (DLF)

Wireless networks are subject to bursty packet losses and frequent topology changes. To provide high routing success rate and low routing stretch even in the presence of frequent topology changes and node/link failures, we develop a simple and effective local failure recovery based on distance vectors.

Overview: A node s retransmits a packet when it does not receive an ACK within a retransmission timeout. When R retransmissions fail, s broadcasts a *failure recovery request*, which contains (i) the next hop s used, (ii) whether destination d is included in s's local cluster, and (iii) the distance to d if s's cluster includes d, or the distance to d's beacon otherwise. Upon hearing the failure requests, s's neighbors attempt to recover the packet locally. Our goal is to select the neighbor that is the closest to the destination as s's new next-hop; meanwhile the selection process should be cheap and easily distributed.

S4 uses distance guided local failure recovery to prioritize neighbors' responses based on their scoped distance vectors. Each node uses its priority to determine the time it needs to wait before sending *failure recovery response*. We further exploit broadcast nature of wireless medium to avoid implosion of recovery responses.

Distance guided local failure recovery: Our goal is to prioritize neighbors based on their distances to the destination so that the nodes closest to the destination can take over the forwarding. The problem is non-trivial, because the distance to the destination is not always available. When the destination is outside the local cluster, a neighbor only knows the distance to the destination's closest beacon, but not the distance from that beacon to the destination.

To address the issues, each node computes its priority using the algorithm in Figure 2. It involves two main scenarios. In the first scenario, s's local cluster contains the destination d. This information is available in s's failure recovery request. Then s's neighbor is assigned one of the four priorities using the following rules. The neighbors that have d in their clusters are assigned the top 3 priorities, since they can directly route towards destination using the shortest path. In this case, each neighbor knows its distance to the destination, and assigns itself a priority based on the difference between $\delta(self, d)$ and $\delta(s, d)$. Neighbors whose local clusters do not contain the destination are assigned the fourth priority, which is the lowest.

In the second case, when s's cluster does not contain the destination d, only the neighbors that have d in their clusters are assigned the highest priority, since they can directly route towards the destination. The other nodes are assigned priorities by comparing their distances to the beacon with $\delta(s, L(d))$.

A sender s selects the neighbor from which it receives the response first as the new next-hop. By assigning each neighbor i with a timer $priority(i) \times m + rand$, a higher priority node sends the response earlier and is thus favored as the new next-hop node. To avoid collisions, we add a small random timer $rand$ to the priority-based timer so that different nodes are likely to respond at different times even when assigned the same priority. To avoid response implosion, upon hearing a failure response to s from someone else, the current node cancels its own pending recovery response if any. Our evaluation uses $m = 50ms$, and $rand$ ranges from 0 to 49ms.

Node failures vs. link failures: The above scheme works well for link failures. When a node fails, all the links to and from the failed nodes are down. Therefore we need to avoid using nodes that use the failed nodes as next hop. This can be done by letting the sender specify the failed node. Only the nodes that use different next hop from the failed node will attempt to recover. In practice, it is difficult to distinguish between a link failure and a node failure. Always assuming a node failure may unnecessarily prune out good next-hops. So we first optimistically assume that the next hop does not fail,

```
// Priorities from highest to lowest: 1, 2, 3, 4
if($d \in C(s)$)
  if($d \in C(self)$) // d is in $s$'s and $self$'s clusters
    $priority = \delta(self, d) - \delta(s, d) + 2$;
  else // d is only in $s$'s cluster
    $priority = 4$;
  end
else if($d \in C(self)$) // $d$ is only in $self$'s cluster
    $priority = 1$;
else // $self$ is outside $s$'s and $d$'s clusters
  $priority = \delta(self, L(d)) - \delta(s, L(d)) + 3$;
end
```

Figure 2: Computing priority using scoped distance vectors and beacon distance vectors

only the link is down. Therefore we allow nodes with the same next hop to recover the packet. When the number of failed attempts pass a threshold, we prevent the nodes from using the same next hop to recover the packet.

3.6 Other Design Issues

Location directory: So far we assume that the source knows which beacon node is closest to the destination. In practice, such information may not be directly available. In such situation, the source can apply the location directory scheme described in BVR [4] to lookup such information. More specifically, beacon nodes are responsible for storing the mapping between non-beacon nodes and their closest beacons. The closest beacon information for node i is stored at $H(i)$, where H is a consistent hash function that maps $nodeid$ to $beaconid$. The source contacts the beacon node whose ID is $H(dest)$ to obtain the closest beacon to $dest$. The storage cost of location directory is much smaller in S4 than that in BVR (as shown in Section 4), because the source in S4 only needs to know the closest beacon to its destination while the source in BVR needs to know the distance between its destination and all beacon nodes. Moreover, in S4 when destination d is in s's cluster, no location lookup is required since s knows the shortest path to d, whereas BVR as well as other geographic routing schemes always require location lookup on a new destination. Such property is especially beneficial when traffic exhibits locality (i.e., nodes close to each other are more likely to communicate).

Beacon maintenance: When a beacon fails, S4 applies distance guided local failure recovery to temporarily route around the failure. If the failure persists, we can apply the beacon maintenance protocol proposed in [4] to select a new beacon. Beacon maintenance is not the focus of this paper. Instead, we focus on the routing performance during the transient period after failures occur.

Link quality: Link quality significantly affects routing performance. We define link quality as the delivery rate of packet on the link in a given direction. In S4, each node continuously monitors its links to/from its neighbors. We adopt a passive link estimator layer developed in [29, 4] for estimating link quality. When a node receives a beacon packet or SDV update, it first checks if *both* the forward and reverse link qualities of the sender are above a threshold (30% is used in our current implementation). Only those updates from a sender with good link quality in both directions will be accepted.

4 Simulation

In this section, we evaluate the efficiency and scalability of S4 by simulation. We compare S4 with BVR [4], because BVR is one of the latest scalable routing protocols and also among the few that have been implemented in real sensor networks. We use BVR with scoped flooding since it provides delivery guarantee and offers a fair baseline comparison. We use three evaluation methodologies: (i) MATLAB simulation based on the unit disk graph radio model (presented in this section), (ii) TOSSIM simulation, a packet-level simulator with more detailed wireless model (presented in Section 5), and (iii) testbed evaluation (presented in Section 6). Our MATLAB simulation results can be directly compared with many previous work on geographic routing, in which the unit disk model is used. TOSSIM simulations allow us to study the performance in more realistic large-scale wireless networks. Having both levels of simulations also reveals how underlying wireless models may affect the routing performance. For BVR, we validate our matlab implementation of BVR by comparing with the original BVR simulation code, and we directly use the original BVR implementation in TinyOS for TOSSIM evaluation.

4.1 Simulation Methodology

To study the protocols in an ideal wireless environment, N nodes are randomly placed in a square rectangle region of size A^2 in the simulator. The packet delivery rates among nodes are derived from the unit disk graph model. That is, each node has a fixed communication range R. A node can communicate with all the nodes inside R, but cannot communicate with any node outside R. It is also assumed that there is no packet loss, collision, or network congestion. In the following description, we let N denote the number of nodes, K denote the number of beacon nodes, R denote communication range, and A^2 denote the size of the area.

We use the following performance metrics to quantify the efficiency and robustness of S4:

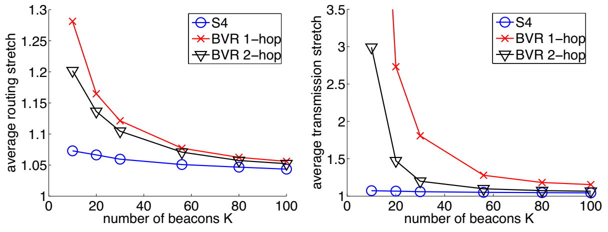
- Routing stretch: the ratio of the route length using the selected routing protocol to that using the optimal shortest path routing protocol.
- Transmission stretch: the ratio of the total number of packets transmitted using the selected routing protocol to that using the optimal shortest path routing protocol.
- Routing state: the amount of state required to maintain at each node.

- Control traffic: the amount of traffic transmitted for setting up the routing state and location directory.

Unless specified otherwise, our default simulation scenario uses a 3200-node network with nodes uniformly distributed in an area of 25×25 square units. The communication range R is 1 unit. On average each node has 15.4 immediate neighbors. Beacon nodes are randomly selected. In BVR, all or a subset of beacon nodes serve as *routing beacons*; a node's coordinate is defined as its distances to the routing beacons. The number of routing beacons K_R is fixed to 10 for all simulations, because it is reported to offer a good balance between routing performance and overhead [4]. For each configuration, we conduct 10 random runs and report the aggregate statistics.

4.2 Simulation Results

4.2.1 Varying the number of beacons (K):



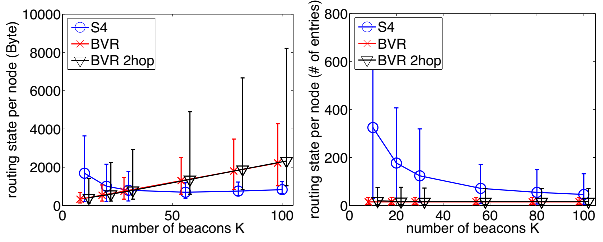
(a) Routing stretch (b) Transmission stretch

Figure 3: S4 has routing and transmission stretches close to 1, which is consistently smaller than those of BVR algorithms across all numbers of beacons.

Routing and transmission stretches: First we compare the routing and transmission stretches of S4 and two variants of BVR by varying the number of beacons K. BVR 1-hop refers to the default BVR algorithm. BVR 2-hop is an on-demand 2-hop neighbor acquisition. In this approach, when a node cannot use greedy forwarding to make progress, it fetches its 1-hop neighbors' neighbors to its routing table. BVR 2-hop reduces the routing failure rate of BVR 1-hop at the cost of higher routing state and control traffic.

Figure 3(a) compares the routing stretches under S4, BVR 1-hop, and BVR 2-hop. The stretches are computed based on 32,000 routes between randomly selected pairs of nodes. We observe that S4 has the lowest average routing stretch. A closer examination of the simulation results shows that the *worst* stretches in S4 are bounded by 3. This is consistent with the worst-case guarantee provided by S4. In comparison, the average routing stretches in BVR 1-hop and 2-hop are substantially higher especially for small K. Moreover their worst-case routing stretches are even higher (*e.g.*, the worst routing stretch of BVR 1-hop in the simulation is 6 for $K = 56$, and much larger for smaller K).

Figure 3(b) compares transmission stretch among the three routing protocols. The average transmission stretches of S4 are consistently below 1.1 under all values of K. However, both BVR 1-hop and BVR 2-hop have much higher stretches when K is small. To achieve comparable transmission stretches to S4 (though still higher), the least numbers of beacons required is 56 for BVR 1-hop and 30 for BVR 2-hop. Such high transmission stretch in BVR is due to its scoped flooding, which is necessary for its guaranteed delivery.



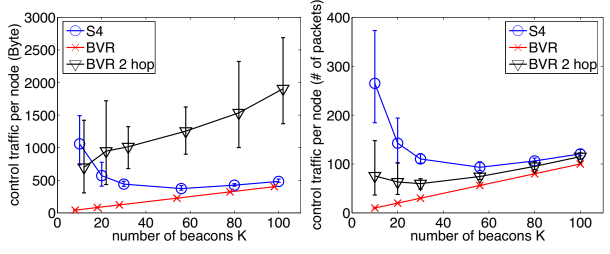
(a) # bytes (b) # routing table entries

Figure 4: Routing state comparison: When $K = \sqrt{N}$, the routing state in S4 is half of routing state in BVR.

Routing state: Figure 4 compares routing state per node under the three routing protocols. The routing state in S4 include route entries for beacon nodes and for nodes within local clusters, whereas the routing state in BVR are determined by the number of neighbors and the length of their beacon vectors K. [2] We make the following observations. First, in BVR the average routing table size proportionally increases with the number of beacons, while the number of entries remains close to the number of neighbors. In comparison, the routing state in S4 first decreases and then slightly increases with the number of beacon nodes. The routing state in S4 reaches minimum for $K \approx \sqrt{N}$ since it gives a good balance between global routing state (for beacon nodes) and local routing state (for nodes in the clusters). These trends also hold for maximum routing state in BVR and S4. Second, recall that to achieve a relatively small transmission stretch, 56 beacon nodes are required in BVR. In this case, the average and maximum routing state in BVR is twice or more than those of S4. Third, BVR 2-hop has significantly higher upper bound of routing state than BVR 1-hop due to the requirement of holding 2-hop neighbor information.

Control traffic: Figure 5 shows initial control traffic for setting up routing state. The bandwidth overhead of BVR 1-hop increases linearly with the number of beacons, because the main overhead is the beacon flooding messages. In BVR 2-hop, other than beacon flooding, the control traffic also includes the overhead of fetching

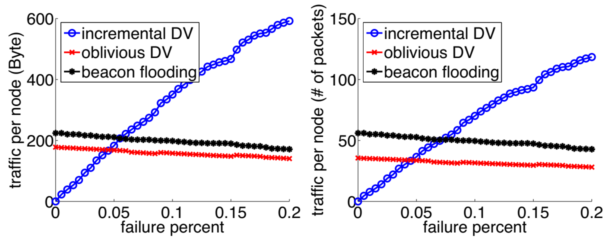
[2] The size of a routing table entry in S4 is 5-byte long in our implementation. The routing state of BVR is estimated based on the relevant data structures found in the BVR implementation code.



(a) # bytes (b) # packets

Figure 5: Initial control traffic to set up routing state: the errorbars show minimum, mean, and maximum traffic across all nodes. The control traffic of S4 decreases gracefully as the number of beacons increases. When $K = \sqrt{N}$, the overhead of S4 is 65% higher than that of BVR 1-hop, but much less than BVR 2-hop.

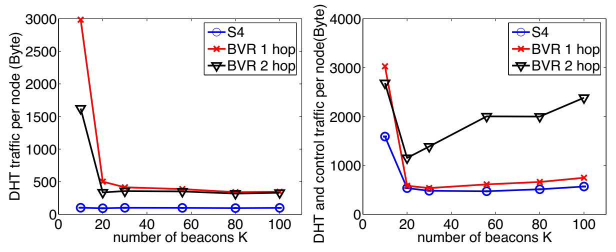
2-hop neighbor coordinates for the required nodes. We can see the overhead of on-demand 2-hop neighbor acquisition is significant, which is a big disadvantage of BVR 2-hop even though its routing stretch is lower than BVR 1-hop. In S4, control traffic includes beacon flooding and SDV. As K increases, the size of the local cluster of each node decreases, so the number of scoped DV packets is reduced. When $K = 56$, the overhead of S4 is 65% higher than that of BVR 1-hop. However since SDV can be updated incrementally after the initial setup, its amortized overhead over the long run is reduced. In terms of the number of packets, S4 is less than twice of the BVR 1-hop when $K \geq \sqrt{N}$. Note that the number of packets in S4 can be reduced by grouping SDV packets. On the other hand, BVR demands large packet size when the number of beacons is large, and large packets could be forced to split in order to achieve high delivery rates under unreliable links.



(a) # bytes (b) # packets

Figure 6: Control traffic overhead of updating routing state due to topology changes

To evaluate the overhead of incremental SDV in S4, we randomly select non-beacon nodes to fail between two consecutive routing updates to create topology changes. There are two ways of updating the routing state after the initial round: either incrementally update based on the current routing state (incremental DV), or builds new routing tables starting from scratch (regular DV). As shown in Figure 6, when the number of node failures is small (*e.g.*, within 5%), incremental routing updates incur lower overhead. Since the typical number of node failures between consecutive routing updates is likely to be low, incremental routing updates are useful in real networks.



(a) Location directory setup traffic (b) Overall control traffic

Figure 7: Control traffic overhead comparison

The control traffic to set up the routing table is not the only overhead. The source should be able to lookup the location information of the destination. Therefore, each node should store its location to a directory during the setup phase. We study such directory setup overhead by using the location directory scheme described in 3.6: each node v periodically publishes its location to a beacon node b_v by using a consistent hashing mechanism. b_v then sends a confirmation back to v if the publishing is successful. We simulate the initial directory setup overhead, in which every node publishes its location to the distributed directory. The results are shown in Figure 7 (a), and they include traffic to and from beacon nodes for publishing the locations. S4 has the following three advantages over the BVR. First, the size of location information in S4 is significantly smaller than that of BVR, because in BVR a node's coordinate is proportional to the number of beacons, while in S4 a node's coordinate is its closest beacon ID. Second, the transmission stretch of BVR is higher than that of S4. Therefore, it incurs more traffic in routing a confirmation packet from the beacon node back to the node publishing its location. Third, it is more likely that a node changes its coordinates in BVR than it changes its closest beacon in S4. Therefore, S4 incurs a lower overhead in setting up and maintaining the location directory.

Figure 7(b) shows the overall traffic overhead incurred in setting up both routing state and directory. We observe that compared with both variants of BVR, S4 has smaller overall control traffic, including traffic in setting up both route and location directory.

Per data packet header overhead: Aside from the control traffic, routing protocols also have overhead in the data packet headers. The overhead of S4 includes the closest beacon ID to the destination and its distance. For BVR, the overhead mainly depends on the number of routing beacons K_R. The packet header of BVR includes a K_R-long destination coordinate, which has at least $\lceil \log_2 \binom{K}{K_R} \rceil$ bits indicating which K_R nodes are chosen out of the total K beacons as the routing beacons for the destination. For example, a rough estimation suggests that with $K = 56$ and $K_R = 10$, BVR requires 15-byte packet headers, which is significant compared to the default packet payload size of 29 bytes in mica2 motes, while S4 only takes 3 bytes in the packet header.

4.2.2 Under obstacles:

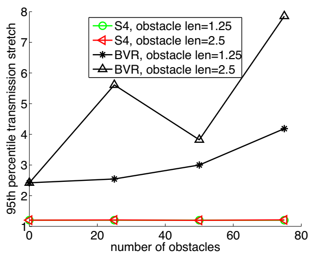


Figure 8: Transmission stretch comparison between S4 and BVR in the presence of obstacles.

We now study the performance of S4 and BVR in the presence of obstacles using the same methodology as in [4]. The obstacles are modeled as horizontal or vertical walls, which completely block wireless signals. (They do not reflect wireless signals.) We vary the number and length of those randomly placed obstacles. We find that the median transmission stretches of S4 and BVR are 1.00 and 1.04, respectively. They are both insensitive to the obstacles. However, as shown in Figure 8, the 95th percentile of the transmission stretches of S4 and BVR are quite different: S4 has a constant 95th percentile stretch around 1.2 regardless the existence of obstacles, while the transmission stretch of BVR increases with the number of the obstacles and the length of the obstacles. For example, when there are 75 obstacles with length 2.5 times of the transmission range, 12.9% of the links are blocked by them. As a result, the 95th percentile transmission stretch of BVR increases up to 7.9 due to the irregular topology, while the stretch of S4 stays around 1.2. This is because S4's worst-case routing stretch guarantee is independent of network topologies.

4.2.3 Summary

Our evaluation shows that S4 provides a worst-case routing stretch of 3 and an average routing stretch around 1.1 - 1.2 in all evaluation scenarios. When $K = \sqrt{N}$ (a favorable operating point for both S4 and BVR), S4 has significantly smaller routing state than BVR. While the initial route setup traffic in S4 is higher than that of BVR, due to its compact location representation, its total control traffic including location setup is still comparable to that of BVR. Furthermore S4 can efficiently adapt to small topology changes using incremental routing update. Finally, BVR 1-hop is more scalable than BVR 2-hop due to its lower control traffic and routing state. So in the following evaluation, we only consider BVR 1-hop as a baseline comparison.

5 TOSSIM Evaluation

We have implemented a prototype of S4 in nesC language for TinyOS [9]. The implementation can be directly used both in TOSSIM simulator [18] and on real sensor motes. In this section, we evaluate the performance of S4 using extensive TOSSIM packet-level simulations. By taking into account actual packet transmissions, collisions, and losses, TOSSIM simulation results are more realistic.

Our evaluation considers a wide range of scenarios by varying the number of beacon nodes, network sizes, network densities, link loss rates, and traffic demands. More specifically, we consider two types of network densities: a high density with an average node degree of 16.6 and a low density with an average node degree of 7.6. We use both lossless links and lossy links that are generated by *LossyBuilder* in TOSSIM. Note that even when links are lossless, packets are still subject to collision losses. In addition, we examine two types of traffic: a single flow and 5 concurrent flows. The request rate is one flow per second for single-flow traffic, and 5 flows per second for 5-flow traffic. The simulation lasts for 1000 seconds. So the total number of routing requests is 1000 for single-flow traffic, and 5000 for 5-flow traffic. We compare S4 with BVR, whose implementation is available from the public CVS repository of TinyOS.

5.1 Routing Performance

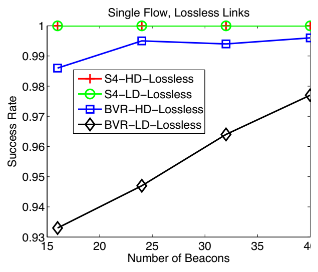
First we compare S4 with BVR in stable networks. To achieve stable networks, we let each node periodically broadcast RBDV and SDV packets every 10 seconds. Data traffic is injected into the network only after route setup is complete. BVR uses scoped flooding after a packet falls back to the closest beacon to the destination and greedy forwarding still fails, whereas S4 uses the distance guided failure recovery scheme to recover failures. To make a fair comparison, in both BVR and S4 beacon nodes periodically broadcast and build spanning trees, and RBDV is turned off in S4.

5.1.1 Varying the number of beacons

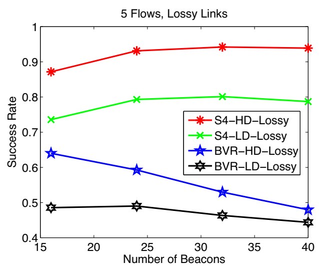
We vary the number of beacon nodes from 16 to 40 while fixing the total number of nodes to 1000.

Routing success rate: We study 4 configurations: a single flow with lossless links, a single flow with lossy links, 5 flows with lossless links, and 5 flows with lossy links. In the interest of space, Figure 9 only shows the results of the first and last configurations. "HD" and "LD" curves represent results under high and low network densities, respectively.

We make the following observations. First, under lossless links with 1 flow, S4 always achieves 100% success



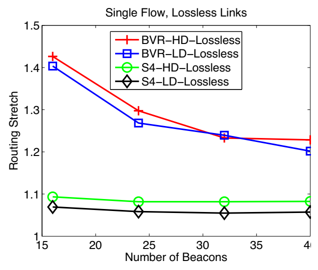
(a) Lossless links w/ 1 flow



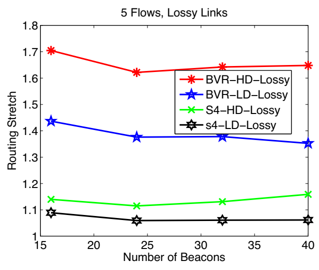
(b) Lossy links w/ 5 flows

Figure 9: Compare routing success under different numbers of beacons, network densities and traffic patterns.

rate. In comparison, BVR achieves close to 100% success only in high-density networks, but its success rate reduces to 93% under low network density with 16 beacons. Why BVR does not provide delivery guarantee even under perfect channel condition? After a packet is stuck at the fallback beacon, scoped flooding is used, which could cause packet collisions and reduce packet delivery rate. Second, under lossy links with 5 flows, packet losses are common, and the performance of both S4 and BVR degrades. Nevertheless, S4 still achieves around 95% routing success rate in high-density networks, while success rate of BVR drops dramatically. The large drop in BVR is because its scoped flooding uses broadcast packets, which have no reliability support from MAC layer; in comparison, data packets are transmitted in unicast under S4, and benefit from link layer retransmissions. Third, the success rate is lowest under low-density networks, with lossy links and 5 flows. Even in this case S4 achieves 70% - 80% success rate, while the success rate of BVR is reduced to below 50%.



(a) Lossless links w/ 1 flow

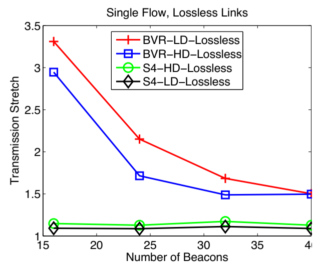


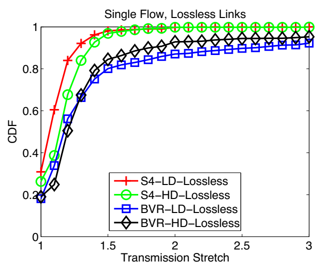
(b) Lossy links w/ 5 flows

Figure 10: Compare routing stretch under different numbers of beacons, network densities, and traffic patterns.

Routing stretch: Figure 10 compares the average routing stretch of S4 and BVR. The average routing stretch is computed only for the packets that have been successfully delivered. Although the worst stretch of S4 is 3, its average stretch is only around 1.1 - 1.2 in all cases. In comparison, BVR has significantly larger routing stretch: its average routing stretch is 1.2 - 1.4 for 1 flow, and 1.4 - 1.7 for 5 flows. Moreover its worst routing stretch (not shown) is 8.

Transmission Stretch: As shown in Figure 11(a), the transmission stretch of S4 is close to its routing stretch, while the transmission stretch of BVR is much larger than its routing stretch due to its scoped flooding. Figure 11(b) shows CDF of transmission stretches under 32 beacon nodes. We observe that the worst-case transmission stretch in S4 is 3, and most of the packets have transmission stretch very close to 1.

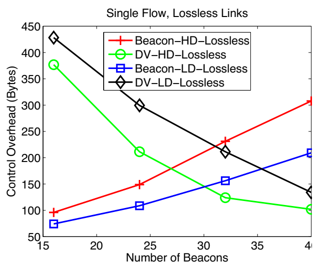


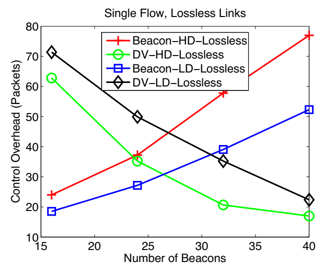


(a) Average transmission stretch (b) CDF of transmission stretch

Figure 11: Transmission stretch comparison

Control traffic overhead: Compared with BVR, S4 introduces extra control traffic of SDV to construct routing tables for local clusters. To evaluate this overhead, we count the average control traffic (in bytes and number of packets) that each node generates under lossless links and a single flow. We separate the global beacon traffic and local SDV traffic. The results are shown in Figure 12. Note that beacon traffic overhead is the same for both S4 and BVR.



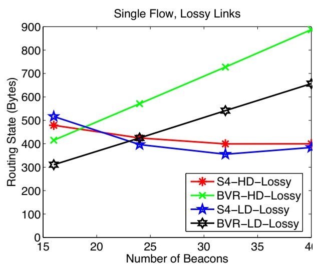


(a) Control traffic in Bytes (b) Control traffic in number of packets

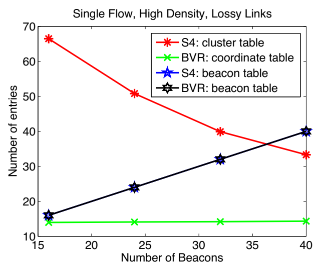
Figure 12: Control traffic overhead under different numbers of beacons and network densities

We can see that when the number of beacons is small, the SDV traffic dominates, since the cluster sizes are relatively large in such case. As the number of beacons increases, the amount of SDV traffic decreases significantly. In particular, when there are 32 beacons ($\approx \sqrt{1000}$), the amount of SDV traffic is comparable to the amount of global beacon traffic. Moreover, if we include control traffic for setting up location directory, the total control traffic in S4 would be comparable to that of BVR, as shown in Figure 7.

Routing state: We compare routing state of S4 and BVR as follows. For S4, the routing state consists of a beacon routing table and a local cluster table. For BVR, the routing state consists of a beacon routing table and a neighbor coordinate table. We first compare the total amount of routing state in bytes between S4 and BVR.



(a) Average routing state



(b) number of routing table entries

Figure 13: Routing state comparison under different numbers of beacons and network densities with lossy links (single flow)

Figure 13(a) shows the average routing state over all nodes. We make the following observations. First, network density has little impact on the routing state of S4, but has large impact on BVR. This is because in S4 the local cluster sizes are not sensitive to network density, while in BVR each node stores the coordinates of its neighbors and its routing state increases with density. Second, the amount of routing state in BVR increases with the number of beacons. In comparison, S4's routing state does not necessarily increase with the number of beacons, since increasing the number of beacons reduces the local cluster size. Third, when the number of beacons is 32 ($\approx \sqrt{1000}$) or above, the routing state in S4 is less than BVR. Similar results have been observed in other TOSSIM configurations as well as MATLAB simulation results in Section 4.

Figure 13(b) further shows the number of entries in beacon routing table, local cluster table and neighbor coordinate table. The beacon table curves of S4 and BVR overlap, since it is common for both. Note that although the coordinate tables in BVR have fewer entries than the cluster tables in S4, the total size of the coordinate tables are generally larger since each coordinate table entry is proportional to the number of beacons.

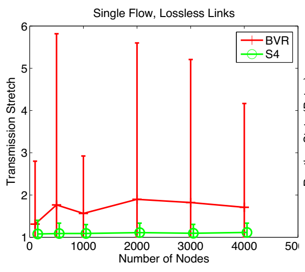
Table 1 shows maximum routing state of S4 and BVR under high density and low density. The maximum number of routing entries is around 4.5 times of $\sqrt{1000}$ (the expected average cluster size), but still an order of magnitude smaller than 1000 (the flat routing table size) in shortest path routing. This suggests that random beacon selection does a reasonable job in limiting worst-case storage cost.

	max S4 state (B)	max BVR state (B)	max S4 routing entries
HD	680	960	136
LD	715	920	143

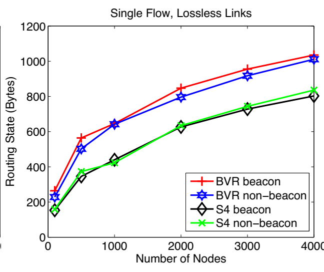
Table 1: Maximum routing state of S4 and BVR

5.1.2 Varying network size

We also evaluate the performance and scalability of S4 when the network size is varied from 100 to 4000. For each network size N, we select $K \approx \sqrt{N}$ nodes as beacon nodes. In the interest of space, we only present results under lossless links and a single flow.



(a) Transmission stretch



(b) Routing state

Figure 14: Comparison under different network sizes

Figure 14(a) shows the average transmission stretch of S4 and BVR under different network sizes. The error bars represent 5- and 95- percentiles. S4 achieves smaller transmission stretches and smaller variations in the stretches. In BVR, packets experience higher medium stretch and higher stretch variation due to greedy forwarding and scoped flooding.

Figure 14(b) shows the average routing state. For both S4 and BVR, the routing state tends to increase with $O(\sqrt{N})$. This suggests both S4 and BVR are scalable with network sizes. In particular, even when the network size is 4000, majority of nodes can store the routing state in a small portion of a 4KB RAM (the RAM size on Mica2 motes we experimented with). Moreover, S4 uses less routing state than BVR when the number of beacon nodes is $\sqrt{N}$, because the coordinate table size in BVR is linear to the number of beacon nodes.

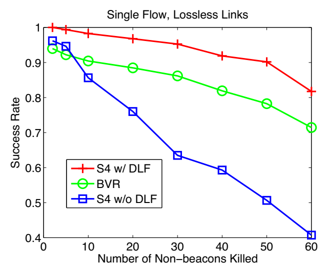
	success rate	routing stretch	transmission stretch	control traffic (B)	routing state (B)
S4	1	1.07	1.08	96	158
BVR	0.994	1.20	1.31	46	232

Table 2: Performance comparison in 100-node networks.

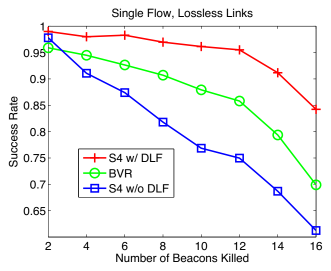
To further study the performance of S4 in smaller networks, we compare S4 and BVR in networks of 100 nodes. Due to space limitation, we only include the results for the case of single flow traffic with lossless links. Table 2 shows that in 100-node networks S4 outperforms BVR in terms of routing success rate, routing stretch, transmission stretch, and routing state. S4 incurs more control overhead of BVR due to the extra SDV traffic, though its overall control traffic (after including location directory setup traffic) is still comparable to that of BVR.

5.2 Impact of Node Failures

To evaluate the performance of S4 under node failures, we randomly kill a certain number of nodes right after the control traffic is finished. We distinguish between beacon and non-beacon failures, and show the results under lossless links and single flow traffic in comparison with BVR. By default, scoped flooding is enabled in BVR.



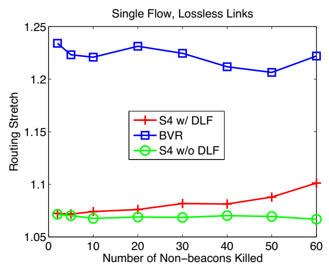
(a) Random non-beacon failures



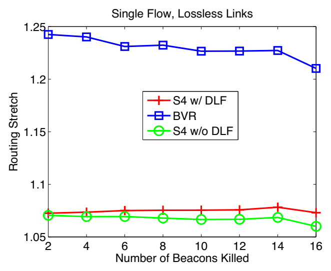
(b) Random beacon failures

Figure 15: Impact of DLF on success rate (1000 nodes, 32 beacons, low density)

Figure 15 shows that failure recovery can significantly increase the success rate under both non-beacon and beacon failures. DLF in S4 is more effective than the scoped flooding in BVR for the following reasons. First, scoped flooding results in packet collisions. Second, S4 uses unicast for data transmissions and benefits from link layer retransmissions. Third, if some node between the beacon and destination fails, DLF can recover such failures, while scoped flooding cannot.



(a) Random non-beacon failures



(b) Random beacon failures

Figure 16: Impact of DLF on routing stretch (1000 nodes, 32 beacons, low density)

Next we compute the average routing stretch over all successfully delivered packets. As we expect, packets going through failure recovery take longer than normal paths. Interestingly, as shown in Figure 16, the average routing stretch is only slightly higher than the case of no failure recovery, which indicates the robustness of S4.

5.3 Summary

Our TOSSIM evaluation further confirms that S4 is scalable to large networks: the average routing state scales with $O(\sqrt{N})$ in an N-node network. The routing and transmission stretches in S4 is around 1.1-1.2. This is true not only in lossless networks under single flow traffic, but also under lossy wireless medium, packet collisions arising from multiple flows, and significant failures. This demonstrates that S4 is efficient and resilient. In comparison, the performance of BVR is sensitive to wireless channel condition. Even under loss-free networks, it may not provide 100% delivery guarantee due to possible packet collisions incurred in scoped flooding. Its routing and transmission stretches also increase with wireless losses and failures.

6 Testbed Evaluation

We have deployed the S4 prototype on a testbed of 42 $mica2$ motes with 915MHz radios on the fifth floor of ACES building at UT Austin. While the testbed is only moderate size and cannot stress test the scalability of S4, it does allow us to evaluate S4 under realistic radio characteristics and failures. We adjust the transmission power to -17dBm for all control and data traffic to obtain an interesting multi-hop topology. With such a power level, the testbed has a network diameter of around 4 to 6 hops, depending on the wireless link quality. 11 motes are connected to the MIB600 Ethernet boards that we use for logging information. They also serve as gateway nodes to forward commands and responses for the remaining 31 battery-powered motes. [3]

We measure packet delivery rates by sending broadcast packets on each mote one by one. Two motes have a link if the delivery rates on both directions are above 30%. Because no two nodes will broadcast packets at the same time, the measurement result is optimistic in the sense that channel contention and network congestion is not considered. The average node degree is 8.7. We observe that a short geographic distance between two motes does not necessarily lead to good link quality. Some of the links are very asymmetric and their qualities vary dramatically over time. Such link characteristics allow us to stress test the performance and resilience of S4.

6.1 Routing Performance

We randomly preselect 6 nodes out of 42 nodes as beacon nodes for S4. The distance from any node to its closest beacon is at most 2 hops. After 10 minutes of booting up all the motes, we randomly select source and destination pairs to evaluate routing performance. The sources are selected from all 42 motes and the destinations are selected from the 11 motes that are connected to the Ethernet boards. All destinations dump the packet delivery confirmation through UART to the PC for further analysis. For each routing request, unless the source is connected to an Ethernet board, we choose the gateway mote that is the closest to the source to forward a command packet. The command packet is sent with

[3]Unfortunately, we are unable to compare S4 against BVR in our testbed. Current BVR implementation requires all motes have Ethernet boards connected to send and receive routing commands. However our testbed only has 11 motes with Ethernet connections, which would make the evaluation less interesting.

time period	# pkts per sec	routing success rate
0 - 70.1 min	1	99.9%
70.1 - 130.2 min	2	99.1%

Table 3: Routing success rate in the 42-node testbed.

the maximum power level, and up to 5 retransmissions so that the source is very likely to receive it. Upon receiving the routing request, the source will send back a response packet with the maximum power level and potential retransmissions, to acknowledge successful reception of the routing request. Each routing request is tagged with a unique sequence number to make the operation idempotent. The data packet will be sent (with the reduced power level) after the command traffic to avoid interference.

We send routing requests at 1 packet per second for the first 70 minutes (altogether 4210 packets), and then double the sending rate thereafter for another 60 minutes (altogether 7701 packets). As shown in Table 3, the routing success rate is 99.1-99.9%, and consistent over time. This demonstrates the resilience of S4 in a real testbed.

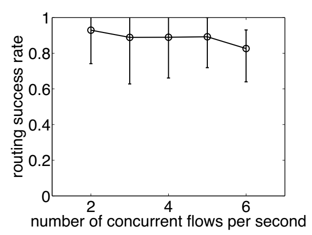
Next we use multiple constant bit rate (CBR) flows to increase the network load. In each multiple flow test, we randomly pick n source destination pairs, and instrument the sources to send consecutive packets at the rate of 1 packet per s seconds. This is essentially having n/s random flows per second. The flows start after a predefined idle period to avoid potential collisions with the command traffic. We choose $s = 2$, and test up to 6 concurrent flows (*i.e.*, n is up to 12). For each experiment, we repeat it for 10 times. Figure 17(a) plots the median routing success rates in different flow settings. The error bars indicate the best-case and worst-case routing success rate. We see the median success rate gracefully degrades with an increasing number of concurrent flows. Our log collected from the gateway motes indicates that some of the failures are due to the limitation of single forwarding buffer per node. Such failure happens when two or more flows try to concurrently route through the same node. Note that this is not a protocol limitation in S4. We could remove many such failures by having a more complete implementation with multiple forwarding buffers, which will be part of our future work.

Finally we study the routing efficiency of S4. Note that it is impossible to calculate the true routing stretch in a real wireless network because the topology is always changing and the packet loss rates depend on the traffic pattern so that the optimal routes are changing, too. Instead, we compare S4 against the *pseudo optimal hop count* metric. The pseudo optimal hop count of a route is defined as the shortest path length in a *snapshot* of the network topology. In our experiment, we use broadcast-based active measurement to obtain the pairwise packet delivery rates before the routing test starts. The delivery rates are averaged over 1-hour measurement period. Note that the real optimal routes could be either better or worse than the pseudo optimal ones due to topology changes, and the delivery rates tend to be optimistic due to no packet collision in the measurement. The routing tests follow the measurement within 30 minutes. We randomly select source and destination pairs and send routing requests at 1 packet per second for 5000 seconds. Then we change the number of beacons from 6 to 3, and repeat the same test. The shortest paths from the topology snapshot are computed offline. Figure 17(b) shows that more than 95% of the routes are within 1-hop difference from the pseudo optimal hops under 6 beacons. Interestingly, S4 sometimes achieves better performance than the pseudo optimal scheme. This is because during the 5000-second routing experiment, S4 adapts to the change of topology so that it can take advantages of new links and reduce path lengths. The number of beacons also has both positive and negative effects on routing performance. When fewer beacons are selected, the nodes tend to have larger routing tables so that more nodes can be reached via the shortest paths; however, having fewer beacons also leads to more control traffic so that the link estimator will have a more pessimistic estimation on link quality due to packet collision. Underestimating link quality apparently hurts the routing performance.

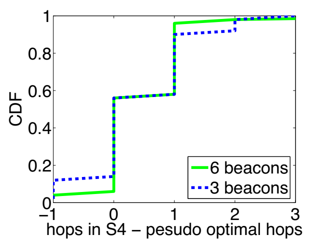
In the same experiment, we also study the routing state per node in S4. Figure 17(c) compares the numbers of local routing table entries used under 6 and 3 beacons. Using 6 beacons yields smaller routing tables. A node in S4 has local routing state towards its neighbor unless the neighbor is a beacon node. Therefore the number of routing entries at each node is generally larger than the number of its neighbors. We find that on average, when 6 beacons are used, the routing table has only 3 more entries than a typical neighborhood table, which suggests that the routing state in S4 is small.

6.2 Routing Under Node Failures

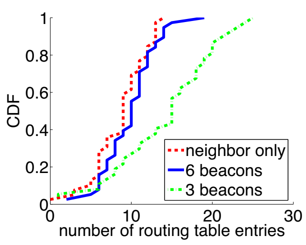
To stress test the resilience of S4, we artificially introduce node failure in our testbed. We randomly select non-gateway motes to kill one by one, and study the routing performance. We send one routing request per second for 50 minutes, altogether generating 3000 packets. The source node is randomly selected from the current live nodes and the destination is one of the gateway motes. Note that we do not start any SDV update or beacon broadcast after the initial setup stage in order to study the effectiveness of the failure recovery mechanism alone. As shown in Figure 17(d), in the first 30 minutes, even when 20 motes are killed, including a beacon node, the routing success rate is still close to 100%. The routing success rate starts to drop after 30 minutes, due to congestion at some bottleneck links. When the



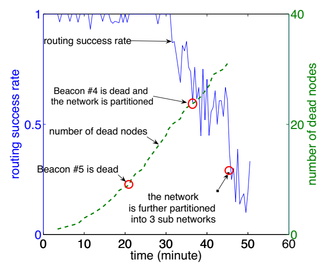
(a) Routing success rate under multiple concurrent flows



(b) CDF of the hop count difference to pseudo optimal



(c) Routing table size



(d) Routing performance under node failure

Figure 17: Experiments on the 42-node testbed

second beacon is killed, the network is partitioned and more routing failures are expected. The third major performance degradation occurs after all 31 non-gateway motes are dead, which causes further network partitions. These results show that S4 is resilient to failures.

7 Conclusion

We present S4 as a scalable routing protocol in large wireless networks to simultaneously minimize routing state and routing stretch in both normal conditions and under node or link failures. S4 incorporates a scoped distance vector protocol (SDV) for intra-cluster routing, a resilient beacon distance vector protocol (RBDV) for inter-cluster routing, and distance-guided local failure recovery (DLF) for achieving resilience under failures and topology changes. S4 uses small amounts of routing state to achieve a worst-case routing stretch of 3 and an average routing stretch of close to 1. Evaluation across a wide range of scenarios, using high-level and packet-level simulators, and real testbed deployment show that S4 achieves scalability, efficiency, and resilience.

Acknowledgement

Thank Deepak Ganesan, Ratul Mahajan, and the anonymous reviewers for their helpful comments on the earlier versions of this paper. We thank Rodrigo Fonseca and Sylvia Ratnasamy for sharing their BVR source code, and thank Yin Zhang for the helpful discussion. This research is sponsored in part by National Science Foundation CNS-0546755, ANI-0319168, CNS-0434515, and Olga and Alberico Pompa Professorship of Engineering and Applied Science at University of Pennsylvania.

References

[1] P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad-hoc wireless networks. In *Proc. of DIALM*, Aug 1999.

[2] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: Network routing inspired by DHTs. In *Proc. of ACM SIGCOMM*, Sept. 2006.

[3] L. Cowen. Compact routing with minimum stretch. *J. of Algorithms*, 2001.

[4] R. Fonseca, S. Ratnasamy, J. Zhao, C. T. Ee, D. Culler, S. Shenker, and I. Stoica. Beacon Vector Routing: Scalable Point-to-Point Routing in Wireless Sensornets. In *Proc. of NSDI'05*, May 2005.

[5] H. Frey and I. Stojmenovic. On delivery guarantees of face and combined greedy-face routing in ad hoc and sensor networks. In *Proc. of MOBICOM 2006*, Sept. 2006.

[6] K. Gabriel and R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, pages 259–278, 1969.

[7] M. Gerla, X. Hong, and G. Pei. Landmark routing for large ad hoc wireless networks. In *Proc. of Globecom*, Nov. 2000.

[8] Z. J. Haas, M. R. Pearlman, and P. Samar. The zone routing protocol (ZRP) for ad hoc networks. Internet-draft, IETF MANET Working Group, July 2002.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the ASPLOS*, 2000.

[10] D. Johnson, D. Maltz, and J. Broch. DSR: The dynamic source routing protocol for multihop wireless ad hoc networks. In *Ad Hoc Networking*, 2001.

[11] B. Karp and H. Kung. Greedy perimeter stateless routing for wireless networks. In *Proc. of ACM MOBICOM*, Aug. 2000.

[12] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. Geographic routing made practical. In *Proc. of NSDI'05*, May 2005.

[13] Y.-J. Kim, R. Govindan, B. Karp, and S. Shenker. On the pitfalls of geographic face routing. In *Proc. of DIAL-M-POMC*, 2005.

[14] F. Kuhn, R. Wattenhofer, Y. Zhang, and A. Zollinger. Geometric ad-hoc routing: Of theory and practice. In *Proc. of ACM PODC*, 2003.

[15] F. Kuhn, R. Wattenhofer, and A. Zollinger. Asymptotically optimal geometric mobile ad-hoc routing. In *Proc. of DIALM*, 2002.

[16] B. Leong, B. Liskov, and R. Morris. Geographic routing without planarization. In *Proc. of NSDI'06*, May 2006.

[17] B. Leong, S. Mitr, and B. Liskov. Path vector face routing: Geographic routing with local face information. In *Proc. of IEEE ICNP*, Nov. 2005.

[18] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *Proc. of ACM SenSys*, 2003.

[19] Y. Mao, F. Wang, L. Qiu, S. Lam, and J. Smith. S4: Small state and small stretch routing protocol for large sensor networks. In *University of Texas at Austin, Dept. of Computer Science, Technical Report TR-07-06*, Feb. 2007.

[20] J. Newsome and D. Song. GEM: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In *Proc. of ACM SenSys'03*, Nov. 2003.

[21] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *Proc. of ACM SIGCOMM*, 1994.

[22] C. E. Perkins and E. M. Royer. Ad hoc on-demand distance vector routing. In *Proc. of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, Feb. 1999.

[23] A. Post and D. Johnson. Self-organizing hierarchical routing for scalable ad hoc networking. *Rice CS Technical Report TR04-433*, 2004.

[24] A. Rao, S. Ratnasamy, C. Papadimitriou, S. Shenker, and I. Stoica. Geographic routing without location information. In *Proc. of ACM Mobicom*, Sept. 2003.

[25] M. Thorup and U. Zwick. Approximate distance oracles. In *Proc. of ACM STOC*, 2001.

[26] M. Thorup and U. Zwick. Compact routing schemes. In *Proc. of SPAA*, Jul. 2001.

[27] G. Toussaint. The relative neighborhood graph of a finite planar set. *Pattern Recognition*, pages 261–268, 1980.

[28] P. F. Tsuchiya. The landmark hierarchy: a new hierarchy for routing in very large networks. In *Proc. of SIGCOMM*, 1988.

[29] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multhop routing in sensor networks. In *Proc. of ACM SenSys'03*, Nov. 2003.

A Location-Based Management System for Enterprise Wireless LANs

Ranveer Chandra, Jitendra Padhye, Alec Wolman, Brian Zill
Microsoft Research

Abstract: The physical locations of clients and access points in a wireless LAN may have a large impact on network performance. However, today's WLAN management tools do not provide information about the location of clients apart from which access point they associate with. In this paper, we describe a scalable and easy-to-deploy WLAN management system that includes a self-configuring location estimation engine. Our system has been in operation on one floor of our building for several months. Using our system to observe WLAN usage in our building, we show that information about client locations is crucial for understanding WLAN performance. Although WLAN location systems are a widely studied topic, the novel aspects of our location system primarily relate to ease of deployment. The main contribution of this paper is to show the *utility of office-granularity location* in performing wireless management tasks.

1 Introduction

Wireless LANs (WLANs) are an important part of today's enterprise networks. However, end users typically do not enjoy the same level of service from WLANs that they have come to expect from wired networks. Better tools for managing WLANs are required for improving the reliability and the level of service provided by today's WLANs.

Wireless networks are fundamentally different from wired networks, in that the behavior of the network is location-dependent. Due to the nature of wireless signal propagation, the physical location of both the transmitter and the receiver may have a large influence on the performance observed by end-users. Specifically, the probability of frame loss, and the data rate selected for frame transmission can be impacted by the locations of the transmitter and the receiver.

The need for incorporating location information in WLAN management tools is also reflected in the common questions asked by administrators of WLANs: Is the access point (AP) placement adequate for serving the locations from where my network is most actively used? Are there areas in my network where clients consistently experience poor performance? How does the distance between an AP and a client affect client's performance? Are there areas that have no coverage at all? With answers to these questions, network administrators can take concrete steps to improve the reliability and performance of their networks.

The WLAN management and monitoring systems available today can not satisfactorily answer these questions. The reason is that many of them provide no information about client's location at all [26, 12]. Others [8, 21, 18] simply approximate the location of the client with the location of the AP that the client is associated with. Our data shows that in our network, 25% of active clients do not associate with the nearest AP. Consequently, these systems can not accurately characterize the influence of location on client performance.

We note that there has been a lot of research on accurate location estimation using WLAN technologies [7, 23, 22, 30, 17]. We believe that the primary reason that these technologies are not integrated with today's WLAN monitoring systems is that the location estimation techniques are generally not easy to deploy. Many location systems require a mapping step whereby an administrator walks throughout the area being covered by the location system to create a "profile" of the environment. Moreover, this profile needs to be updated at regular intervals to ensure that it reflects the current environment.

We have designed and implemented a WLAN management system with an integrated, self-configuring indoor location system. Our location system is accurate to the granularity of individual offices and requires minimal manual intervention to setup and operate.

Our system is built upon the DAIR platform described in [6, 5]. The DAIR platform turns ordinary user desktops into wireless monitors (AirMonitors) by attaching a wireless USB dongle to each of them. The DAIR architecture allows us to create a dense deployment of WLAN monitors in a scalable and cost-effective manner.

The dense deployment of AirMonitors has several advantages. In most cases, there is at least a single AirMonitor that can hear a majority of the packets flowing between a given client and its AP in a single direction (another AirMonitor may hear most of the packets flowing between the client and the AP in the other direction). This allows us to sidestep the complex tasks of trace merging and fine-grained time synchronization faced by other WLAN monitoring systems [26, 12]. The dense deployment also allows us to use very simple location estimation algorithms, yet achieve office-level accuracy.

We have deployed this system in our building over the last six months. Our current deployment consists of 59 AirMonitors, and covers an entire floor. We have been using it to monitor the WLAN in our building. During this time, our system was able to answer many of the questions we posed earlier. For example, we detected that clients in one corner of our building received consistently poor performance. We were able to provide a fine-grained characterization of the workload on our network: we noticed that clients located in people's offices tend to download more data than clients located in various conference

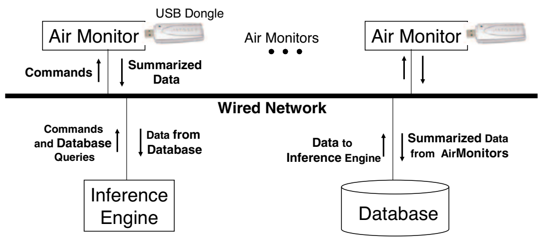


Figure 1: *The DAIR Architecture.*

rooms. We characterized the impact of distance on uplink and downlink transmission rates as well as loss rates in our environment. Much to the delight of system administrators, we also located transmitters that were sending malformed 802.11 packets. We discovered and reported a serious misconfiguration shortly after new APs were deployed on our floor. These APs sent downlink traffic at 5.5Mbps, regardless of the location of the client. This problem has since been fixed.

In summary, the key contributions of our paper are:

- To the best of our knowledge, we are the first to integrate *office-level* location accuracy into a WLAN management system.
- We show that correlating client locations with a variety of performance metrics yields new insights into WLAN behavior.
- We demonstrate the usefulness of our system by using it to monitor an operational WLAN.
- We show that by using a dense deployment of wireless sensors, one can significantly simplify the tasks of wireless monitoring and location estimation.

2 The DAIR Platform

The design and the architecture of the DAIR system has been described in detail in [5]. Here, we provide a brief review of the system architecture.

The DAIR system is designed for easy and inexpensive deployment in enterprise environments. Existing desktop machines serve double-duty as WLAN monitors. The IT department can mandate which desktops perform this service, and they can also manage the process of deploying the DAIR software on these systems.

Figure 1 provides a high-level illustration of the three major components of the DAIR system: the AirMonitors; the database server; and the inference engine. We use the term AirMonitor to refer to ordinary desktop computers in the enterprise that are equipped with inexpensive USB 802.11 wireless cards and have two components of the DAIR software installed: (1) the AirMonitor service; and (2) a custom device driver that works with USB wireless cards based on the Atheros chipset. The AirMonitor service is user-level code that runs as a Windows service, the equivalent of a daemon on Unix systems. The device driver customizations allow the wireless card to receive all 802.11 frames, including those destined for other 802.11 stations and those with decoding errors.

The AirMonitor service contains all of the user-level code for monitoring. It enables packet logging at the driver level, at which point all frames are delivered to the service. Within the service, the basic unit of extensibility is a "filter': each new application built to use the DAIR system installs an application-specific filter that runs inside the AirMonitor service. Each frame from the driver is delivered to all running filters. The filter's primary task is to analyze the frames, summarize them in an application-specific manner, and then submit those summaries to the database server.

The intent is that filters do whatever summarization is sensible to improve the scalability of the system without imposing an undue CPU burden on the AirMonitors – we don't want to submit every frame that each AirMonitor overhears to the database, yet we also don't want the AirMonitors to do all of the complex data analysis, which is the responsibility of the inference engine. While the precise definition of what constitutes undue burden varies based on circumstances, parameters such as history of CPU and memory usage are taken into consideration [14].

We use Microsoft's SQL Server 2005 as our database server. We made no custom modifications to the database server. The DAIR system is designed to scale to handle very large enterprises. When the number of AirMonitors in the system exceeds the capacity of a single database server, one can simply deploy another database server. However, AirMonitors should be assigned to servers in a location-aware manner, to limit the number of queries that must be performed across multiple database servers.

The computationally intensive analysis tasks are all performed by the inference engines. Inference engines are stand-alone programs that analyze the data gathered by the AirMonitors. The inference engines learn about new events by issuing periodic queries to the database server.

3 Management System Design

In this section, we describe the new infrastructure components, beyond the original DAIR platform described in our previous work, that are utilized by all of our wireless management applications.

3.1 Location engine

The goal of our location engine is to determine the location of any 802.11-compatible transmitter (which includes WLAN clients such as laptops and hand-held devices) on our office floor, and to communicate that location to the rest of the management system. Our design was guided by the following set of requirements.

First, we require no cooperation from the clients: no special software or hardware is needed, and the clients need not communicate directly with the location system.

Second, the location system should provide "office-level" accuracy: the error should be within 3 meters, approximately the size of a typical office. Although other proposed location systems provide greater accuracy, this level is sufficient for our needs. Third, to ensure easy deployment, the system must be self-configuring – it cannot require manual calibration. Finally, the location system must produce output in a way that is physically meaningful to the network administrators, which precludes having the system construct its own virtual coordinate space as Vivaldi does [13].

The basic idea behind our location system is similar to that of many previous 802.11-based location systems. The AirMonitors record the signal strengths of frames transmitted by a sender. This information is combined with the known AirMonitor locations to estimate the location of the transmitter. The key distinguishing features of our location system are: 1) by deploying AirMonitors with high density, we can avoid the manual profiling step required by previous indoor 802.11 location systems [7, 17]; and 2) we use external sources of information commonly available in enterprises environments to automatically determine the location of most AirMonitors.

In the remainder of this section, we describe the bootstrapping method for automatically determining the AirMonitor locations, followed by the three simple location estimation algorithms supported by the location engine.

3.1.1 Determining the AirMonitor Locations

To automatically determine the physical location of the AirMonitors, we start by determining the number of the office that each AirMonitor is located in. Because the DAIR system uses end-user desktops for the AirMonitors, we can analyze the login history of these machines to determine who the primary user is. In a typical enterprise, the occupant of an office is generally the primary user for the machines in that office. We examine the system eventlog for user login and console unlock events, and extract the user identifier for these events. We ignore remote login events and non-console unlock events. We then determine the primary user by selecting the user with the most login and unlock events in a given time period. With this information, we then consult a database of users and office numbers (our implementation uses the Microsoft Active Directory service [27]) to determine the office number where the machine is likely located.

The final step uses online to-scale building maps, which are available to us in Visio XML format. The map includes labels for the office numbers, which are centered within each office. We parse the XML, determine the graphical coordinates of each label, and convert these to the physical coordinates. This procedure gives us an estimate of the center of each office. By combining this information with the user-to-office mapping, we can automatically determine the location of most of our AirMonitors.

The details of this procedure are clearly specific to our particular environment. Although we believe that many enterprise environments maintain similar types of information, the specific applications and data formats may differ. Also, even in our environment these techniques are not completely general. For example, this procedure tends not to work well for machines located in public spaces, such as conference rooms and lounges, because the login history on these machine does not tell us much. For such machines, we still must manually inform the system of the office number to get the coordinates into our database.

Our assumption that the AirMonitor location is in the center of each office is another source of error. We do not try to pinpoint the location of an AirMonitor within an office because doing so would require significant manual effort. This approximation is appropriate for us because we only require office-level accuracy. We assume that the physical location of AirMonitors does not change often. We determine the location of the AirMonitor when we first deploy it and re-confirm it only infrequently.

3.1.2 Locating a Transmitter: StrongestAM

During the course of their normal operation, various AirMonitors hear the frames sent by the transmitter, which we identify by the sender MAC address. On a periodic basis, the AirMonitors submit summaries to the database of the signal strength of those frames overheard. These summaries contain start and end timestamps, the AirMonitor identifier, the sender MAC address, the channel on which the frames were heard, the number of frames sent by this transmitter, and the total RSSI of those frames.

When the inference engine wants to locate a client, it provides the start and end time and the sender MAC address to the location engine. The location engine computes the average signal strength seen by each AirMonitor during the specified time period for that transmitter. Then, it chooses the AirMonitor that saw the highest average RSSI (i.e. strongest signal strength) during this period and reports this AirMonitor's location as the location of the client.

The StrongestAM algorithm is very simple, and is likely to give inaccurate results in many cases. One reason is that even when a client is stationary, the RSSI seen by the AirMonitors can fluctuate significantly. Some of these fluctuations are masked due to averaging over multiple packets. Yet, if two AirMonitors are reporting roughly equal signal strength, the fluctuations may make it impossible to pick the strongest AM without ambiguity.

However, if the AirMonitor density is high enough, this simple algorithm might suffice to provide office-level granularity in most cases. As we will show later, this algorithm works well primarily when the client is located in an office where we do have an AirMonitor.

3.1.3 Locating a Transmitter: Centroid

The next algorithm we implemented, Centroid, is a variant on the StrongestAM: instead of selecting only the strongest AirMonitor, we select all AirMonitors whose average RSSI is within 15% of the average RSSI of the strongest AirMonitor. We then report the client location as the centroid of the set of selected AirMonitors. If there are no AMs within 15% of the strongest AM, the location is just reported to be that of the strongest AM.

This algorithm has several attractive features, especially in our deployment, where we have an AirMonitor in about half of the offices on our floor. When the transmitter is in the same office with an AM, it is generally the case that this AM sees significantly higher RSSI values, compared to all other AMs. As a result, the office is correctly picked as the location of the transmitter, without any adjustment. Also, the accuracy of this algorithm is not affected by the moderate adjustments in transmit power that are seen when clients use transmit power control. If a packet is sent at a lower power level, all the AMs will see a proportional decrease in the observed RSSI.

In addition, the Centroid algorithm is unaffected by small fluctuations in the RSSI that various AirMonitors report. The threshold implies that the set of AMs that we compute the centroid over does not change with small fluctuations in reported signal strength.

The Centroid algorithm does have one drawback. If the set of AirMonitors we compute the centroid over are all on one side of the actual client location, we could see a significant error. As it turns out, this is not a significant problem in our deployment but it certainly could be in other environments. We discuss the issue more in Section 5. To address this problem, we developed a third algorithm which we discuss next.

3.1.4 Locating a Transmitter: Spring-and-ball

The previous two algorithms do not explicitly take into account the radio propagation characteristics of the local environment. Our third algorithm, Spring-and-ball (inspired by Vivaldi [13]), addresses this problem by using "profiles" of the area in which the WLAN is deployed.

A profile is an approximate, compact representation of how the signal strength degrades with distance in the local environment. To generate a profile, each AirMonitor broadcasts special probe packets at regular intervals, that contain the identity of the transmitting AirMonitor. The other AirMonitors record these probe packets and report the average signal strength of packets heard from each AirMonitor to the central database. Using the known distance between two AirMonitors and the observed average RSSI between those AirMonitors, the inference engine fits a set of simple curves to the combined observation data, and picks the best fit as the profile of the environment. The resulting profile is labeled with the channel on which the probe packets were sent and stored in the database.

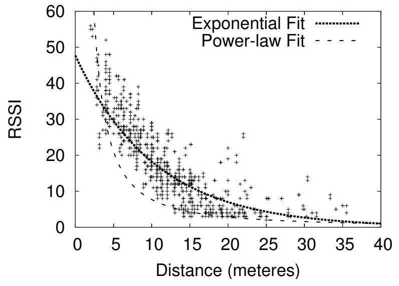


Figure 2: *Sample profile data and two fitted curves. The equation of the exponential curve is:* $RSSI = 47.72 * e^{-0.096*Dist}$*, while the equation of the power-law curve is* $RSSI = 215.73 * Dist^{-1.445}$.

We currently consider linear, exponential, logarithmic, and power law curves. Each of these curves can be fitted using variants of the least-squares fitting method. We do not filter the raw data in any way; all points are considered while determining the fit. The goodness of fit is determined by the R^2 (correlation coefficient) value. An example is shown in Figure 2.

When the location engine is asked to locate a client, it computes the average signal strength that each AirMonitor observed. It then calculates the initial estimate of the location using the Centroid algorithm. The estimate is the refined using a Spring-and-ball algorithm as follows.

We select the most recent profile that matches the frequency band of the channel on which the packets were heard. Using the profile, we calculate the signal strength that each AirMonitor *should* have seen had the transmitter been at the estimated location[1]. We then consider the difference between the calculated signal strength and the signal strength that the AirMonitor actually observed.

The absolute value of the difference corresponds to the magnitude of the force on the "spring" that connects the transmitter to the AirMonitor. The sign of the difference indicates whether the spring is compressed or stretched. The direction of the force is along the line connecting the AirMonitor location to the estimated transmitter location.

We then move the estimated location of the transmitter a short distance in the direction of the cumulative force [2]. This reduces the magnitude of the error by a small amount. This is the new estimated location of the client. We recalculate the forces at this new location, and repeat the process until one of the following is true: (i) 5000 iterations have elapsed, (ii) the magnitude of the error falls below 0.1, or (iii) the force at the new location exceeds by 10% the minimum force seen so far.

[1]If the calculated signal strength value is less than 0, it is assumed to be 0. Similarly, if the calculated signal strength value exceeds 100, it is assumed to be 100.

[2]If either the X or the Y co-ordinate of the location falls outside the boundary of the floor, it is adjusted to be within the boundary.

The Spring-and-ball algorithm will converge to a global minimum only when the magnitude of the force is linearly proportional to the distance [13]. This is not true in our setting, since the drop in signal strengths is not linearly proportional to the distance. However, we have found that the algorithm works well in practice.

This algorithm overcomes the key limitation of the Centroid algorithm. If all the AirMonitors that hear a particular client are on one side, the Centroid algorithm would pull the client to be in midst of those AirMonitors. The Spring-and-ball algorithm will realize that the signal strengths reported by the AMs are too low to be coming from the central location, and will push the client to the correct side.

The key challenge in implementing the Spring-and-ball algorithm is the generation of profiles. Questions that arise include: How often should profiles be generated (i.e. do time-of-day effects create significant errors)? What type of curve should we use to fit the data? We have carried out extensive experimental sensitivity analysis of the Spring-and-ball algorithm. We have found that the time of day effects do not significantly impact the accuracy. In fact, we only generate new profiles when we add new AirMonitors to the system. We have also found that either a log or power curve provides a good fit to the observed data. We do not describe the results of our study further, because we want to focus our results on the *use* of the location system for management applications.

Some of the previous WLAN location estimation systems proposed in the literature are more accurate than our system. Our goal, however, was to design a location system that provided office-level accuracy, and that met our rather stringent requirements for ease of deployment.

3.2 AP Tracking

The corporate APs in our building are controlled by a central server [3], which dynamically reconfigures which channels the APs operate on. The frequency of change depends on a variety of factors, including traffic load. We have observed some of our APs changing channels as many as six times in a 12 hour period. As a result of this dynamic channel assignment, we can not assign our AirMonitors to listen to fixed channels. Instead, each AirMonitor is assigned to track the nearest AP – this configuration is shown in Figure 3.

The AirMonitor implements AP tracking by continuously looking for beacons sent by the assigned AP. If no beacons are observed during a 10-second period, the AirMonitor goes into scanning mode, where it listens for beacons for 150 ms on each of the 11 channels in the 2.4 GHz band, until the assigned AP is heard from again. If scanning fails to find the assigned AP within two minutes, it goes back to the previous channel where it heard the AP, on the assumption that the AP may have failed. Every 30 minutes, it re-enters scanning mode and looks for the AP on the other channels. While in scanning mode, packet delivery to all the filters is suspended.

3.3 Address Matching

The frame format of the 802.11 standard is not fully self-describing. One example of this is that there are two types of 802.11 frames, namely the CTS (clear-to-send) and ACK (acknowledgment) frames, that do not contain the MAC address of the device that transmits those frames. This means that for devices using promiscuous mode to passively listen to 802.11 conversations, they cannot directly determine who sent those packets. Fortunately, if the passive listener is capable of overhearing both directions of the conversation, then it is possible to infer the address of the transmitter of such frames.

One component of the AirMonitor service is responsible for inferring and then filling in both the transmitter and BSSID fields for both CTS and ACK frames. This information is filled in before the frame is handed off to any other filters that are running on the AirMonitor, and therefore these filters can simply assume that this information is available whenever matching was possible.

The strategies we use to infer the transmitter are different for ACK frames than for CTS frames. To determine the transmitter of an ACK frame, we need to analyze packets that arrive back-to-back. If the first frame type is a Data or Management frame, then we remember the value of the Duration header for that frame. When the next frame arrives, we determine if it is an ACK frame that has arrived within the time period allocated by the previous frame. If so, then we know that the receiver of the first frame must be the transmitter of the ACK frame. Furthermore, we use the ToDS and FromDS bits in the header of the previous frame to determine the correct BSSID to attribute to this ACK frame.

To infer the transmitter for CTS frames, we considered using a similar strategy where the previous frame should be an RTS frame. However, in our network the vast majority (more than 99%) of CTS frames are "CTS-to-self" frames that are used for 802.11g-mode protection. In other words, there is no previous RTS frame to match with. Fortunately, the convention for these g-mode protection frames is that the Duration of the CTS frame covers the time period when the actual data frame will be sent, and the receiver address in the CTS frame should match the transmitter address of the following frame.

There is additional complexity in our system for matching packets where the Duration header is empty, as is the case for Data Null frames. Due to space constraints, we provide these details in [11].

3.4 Time Synchronization

The AirMonitors timestamp the data they submit to the central database using their own local clock. We shall see in Sections 4.1 and 4.2 that the inference engine sometimes needs to correlate the data submitted by various AirMonitors. Hence, we need to synchronize the AirMonitor clocks. However, unlike [26, 12] we do not need to correlate data at packet level granularity. As a result, time synchronization provided by NTP [28] is sufficient for our purposes.

4 Management Applications

In this section we explore three location-aware WLAN management applications that our location engine enables. The Client Performance Tracker application monitors the data transfers between clients and APs, and correlates client location with a variety of performance statistics. The Connectivity Tracker monitors the association behavior of clients, and correlates connectivity problems with client location. Finally, the Bad Frame Tracker detects and locates transmitters which send invalid 802.11 frames.

We implement each of these applications with two components: a filter that runs on each AirMonitor and summarizes the overheard frames into the relevant data for that application, and an inference engine which queries the central database to analyze the summarized data.

4.1 The Client Performance Tracker

The goal of the client performance tracker application is to provide insight for the network administrator into where the clients are using the wireless LAN, and to provide aggregate statistics about the nature of their usage and the quality of service they obtain. With information about client locations, this allows the administrator to look at the relationship between a variety of performance metrics and location. There are many different interesting questions one can answer with this application. For example: do active clients typically associate with the nearest AP? What is the relationship between distance from the client to the AP and the observed loss rates? How is transmission-rate selection affected by the distance from the client to the AP? Where are the most heavily utilized locations in service area? Do the APs do a good job of covering those locations?

The Client Performance Filter: The client performance filter submits data summaries to the database using a randomized interval between 30 and 60 seconds. The randomization is used to avoid having all the clients submit in a synchronized manner, thus making the load on the database server more uniform.

For each (transmitter, receiver) pair, the client performance filter maintains a large set of aggregate counters.

Counter	Description
TotalCount	Total number of frames.
TotalBytes	Total number of bytes in all frames.
DataCount	Total number of Data frames, excluding Data Null frames.
DataBytes	Total number of bytes in Data frames, excluding Data Null frames.
DataNullCount	Total number of Data Null frames.
DataNullBytes	Total number of bytes in Data Null frames.
MgmtCount	Total number of Management frames.
MgmtBytes	Total number of bytes in Management frames.
CtrlCount	Total number of Control frames.
CtrlBytes	Total number of bytes in Control frames.
RetryCount	Total number of frames where the Retry bit is set.
RetryBytes	Total number of bytes in Retry frames.

Table 1: *Aggregate counters maintained by the client performance filter on a per (transmitter, receiver) pair basis.*

A complete list of these counters is shown in Figure 4.1. Note that for these counters, address matching has already occurred (see Section 3.3), so the counters include frames for which the transmitter address was inferred. For those frames where the transmitter could not be inferred, the counters will not be incremented. We will see shortly how the inference engine of the client performance tracker attempts to compensate for this.

In addition to the basic usage statistics, the filter collects two additional kinds of information to allow analysis of clients' auto-rate and packet loss behavior. For auto-rate analysis the filter collects two histograms: one of the number of packets transmitted at each of the possible rates, and another of the number of bytes transmitted at each of the rates.

Understanding the packet loss behavior requires the filter to perform a significant amount of online analysis, which we now summarize. The complete details of our loss estimation algorithm can be found in [11]. The techniques we use to analyze uplink (client to AP) and downlink (AP to client) traffic differ. For downlink traffic, we use an approach very similar to the "nitWit" component of the Wit system [26] to infer certain packets were transmitted even though our AirMonitors did not directly observe them. For example, when we see a data frame with a sequence number that we have not seen before, and the retry bit is set on that data frame, then we know that an initial data frame with that same sequence number was transmitted, even though our AirMonitor did not observe it. Additional information is provided by ACKs that travel in the reverse direction, and the address matching code allows us to correctly attribute the transmitter of these ACKs. Our analysis provides us with two *estimated* values: an estimate of the combined number of management and data frames transmitted, as well as an estimate of the loss rate.

For the uplink traffic, in addition we also analyze gaps in the sequence number space to infer more about the number of frames transmitted, as was suggested as future work in the Wit paper. Because stations operating in infrastructure-mode only communicate with one access point at a time, we can analyze gaps in the sequence number space to further infer traffic that our monitor has missed. This has to be done carefully, however. Clients periodically perform scanning: they change channels to search for other nearby access points. Fortunately, most clients send a Data Null frame to the AP with the power-save bit set just before switching channels, so that the AP will buffer any incoming data, and they send another Data Null with power-save bit cleared when they return from scanning. By analyzing these Data Nulls, we avoid attributing these gaps as communication between the station and AP that was missed by our AirMonitor.

One additional issue arises with analyzing sequence number gaps: suppose that during a single summarization period, a client begins by communicating with AP1, then switches to AP2, and later switches back to AP1. The AirMonitor may not see any of the communication with AP2 because it may take place on a different channel. To deal with this situation, we take a conservative approach: we discard the inferred counters for any row where this "AP flipping" behavior is detected, and we use the inference engine to detect this situation after the fact, by analyzing the client association data summarized by the Connectivity Tracker.

The Client Performance Inference Engine: As described above, we may have many different AirMonitors observing and analyzing the same communication between a transmitter and a receiver, and many of the AirMonitors will be performing their analysis based on incomplete views of this communication. Therefore, the key job of the inference engine is to select, during a particular time interval, the two AirMonitors that were best able to overhear each direction of the communication. In other words, we select the best AirMonitor for the uplink, and the best AirMonitor for the downlink, and these may or may not end up being the same AirMonitor. Typically, the Inference engine ends up selecting one of the AirMonitors that is physically close to the transmitter.

We analyze each direction of the communication separately, but in order for our loss estimates to be accurate, the AirMonitor analyzing frames flowing in one direction still needs to be able to overhear the corresponding ACK frames flowing in the opposite direction. Fortunately, we never see ACKs transmitted at rates higher than 24 Mbps, which means that it is typically much easier for AirMonitors to overhear them than it is to overhear high data-rate frames such as those sent at 48 or 54 Mbps. To decide which AirMonitors to select during a given time interval, we simply use packet counts to figure out which AirMonitor was best able to overhear the particular conversation.

Using the AirMonitor selection metric of most packets overheard is potentially in conflict with the randomized data submission strategy used by the client performance filter. In other words, we don't want to always end up being biased toward choosing the AirMonitor that submitted data most recently. To avoid this problem, the inference engine must analyze the data over a timescale that is significantly longer than the data submission interval. We typically use a 15 minute analysis interval for the inference engine, versus a 30 to 60 second submission interval for the filters. The impact of this design choice is that the Client Performance Tracker application cannot provide instantaneous information about performance problems to the network administrator, it can only provide summaries about moderately recent activity.

In Section 5.2, we demonstrate the utility of the Client Performance Tracker by using it to analyze the operational wireless LAN in our building, and by showing the results for our network to many of the questions raised at the beginning of this section.

4.2 The Connectivity Tracker

An 802.11 client goes through a sequence of steps before it can connect to an AP. First, it first sends probes to discover nearby APs, then it authenticates to a selected AP, and finally it associates to that AP. Each of these steps involves a client request frame followed by an AP response frame. Once the client has associated, it can exchange data with the AP.

The connectivity tracker monitors and summarizes this association process for all clients while tracking their locations. Analyzing this data helps a network administrator answer a number of key management questions, such as: How many clients in a particular region were able to connect to the WLAN, and for how long? Are there any regions with no RF coverage (i.e., RF holes)? Is connection duration correlated with location (e.g., people in conference rooms may have short lived connections, while those from offices may be long lived)? Are there specific regions where clients rapidly switch back and forth between APs? Understanding the location of clients is crucial to answering these questions. Also, note that existing AP-based wireless management systems may never even learn of the existence of those clients in an RF hole.

The Connectivity Filter: This filter records the sequence of association steps executed by each client. This information is recorded in local memory as the Connectivity table, and this table is periodically flushed to the central database.

The connectivity table records this information as a sequence of association states, which are tuples containing *(client, Bssid, state, NumFrames, NumAcked)* along with a start and end timestamp. The current association state for each client represents the last non-probe management packet that was exchanged between the client and the AP with the corresponding BSSID. The filter checks every incoming packet to see if it changes the current association state of the client, and if so, the filter generates a new row in the Connectivity table.

Note that certain events may appear in different sequences than the one described above. For example, clients may perform an active scan which involves sending and receiving probes even while maintaining their current association with an AP. Similarly, clients may perform pre-authentication with another AP while associated [19].

The above table only contains information about clients that send or receive at least one packet. It does not have any information about idle clients, or clients that are disconnected and have given up trying to associate to any AP. The latter situation is particularly interesting, and may arise when clients are in an RF hole. To address this, the connectivity filter periodically sends out beacons pretending to be a valid AP, in an attempt to induce any clients in an RF hole to initiate the association process. We mark these special beacons using an Information Element (IE) so that the other AirMonitor filters can ignore these frames, but real clients will not.

The Connectivity Inference Engine: The inference engine queries the central database and aggregates all the information submitted by the connectivity filters on the AirMonitors. This complete view of all association events seen by our system is then sorted by timestamp. We then take advantage of the time synchronization across AirMonitors to coalesce the same event seen by multiple AirMonitors. This step is similar to trace merging [26, 12], but at a coarser granularity. Finally, we perform a post-processing sanity check over the coalesced sequence of events using our knowledge of the 802.11 association procedure, similar in spirit to the analysis that WIT does for data transfers.

We use this final sequence of events to do detailed analysis of client connectivity from various locations. We are able to characterize RF Holes even when communication is possible in only one direction. We can also infer the duration of client associations to an AP, and we can detect regions where clients cannot maintain stable associations, and therefore switch rapidly between APs.

4.3 The Bad Frame Tracker

One common problem facing the administrators of large corporate WLAN networks is to locate non-compliant transmitters. A transmitter may send non-compliant 802.11 frames for many reasons, such as errors in hardware and/or driver software bugs. We see several examples of such malformed frames in our network. Although it is rare, such malformed frames may cause problems for certain receivers. Locating the source of such frames can be helpful in debugging the underlying problem.

We have built a Bad Frame filter that logs only the non-compliant 802.11 frames. Unlike other filters which summarize the data, the Bad Frame filter simply submits the raw contents of bad frames to the database, as long as the frame checksum is correct. The corresponding inference engine attempts to localize these packets, and generates reports for our system administrators.

As one example, on our network we observed approximately 10 to 20 frames per day, heard by multiple AirMonitors, that all followed the same pattern. These frames carried a valid Frame Type (Control), but an invalid SubType. The FromDS and ToDS bits in the frame header were both set to 1, which is disallowed by the 802.11 standard. At first glance, the frame payload offered few clues about their origin. Using our location engine we were able to calculate an estimated location for each frame, and we found that the estimated locations were always near two of the six APs on our floor. On closer inspection of the frame payloads, we realized that a set of 8 bytes in the frame was a nibble-wise rearrangement of the MAC address of the AP near the estimated location. We contacted our IT administrators with this information, and the AP vendor is currently investigating this problem.

This example illustrates that our location engine is able to accurately calculate estimated locations with just a few samples. Without the location information, we would have been unlikely to realize that our APs were the origin of these malformed frames.

5 Experimental Results

Our system is deployed on one floor of a fairly typical office building. Our building has rooms with floor-to-ceiling walls and solid wood doors. There is a corporate wireless LAN with six 802.11 a/b/g access points operating on our floor. The six corporate APs offer service in both the 2.4GHz and the 5Ghz band. There is little traffic on the 5GHz band in our building, so for the purpose of this paper we only monitored the 2.4GHz band. Our current DAIR system deployment consists of 59 AirMonitors as shown in Figure 3, and a single database server. The AirMonitors testbed consists of various types of desktop PCs. Each AirMonitor is equipped with either a Netgear WG111U or a Dlink DWL-AG132 USB wireless dongle.

5.1 System Validation

We validate our system design in three ways. First, we show that our monitoring software and hardware, together with our channel assignment is capable of observing most packets. We also show that the strategy used by the Client Performance Tracker of selecting the best AirMonitor to overhear a particular conversation between a client and an AP does not result in excessive loss of information. Second, we evaluate the accuracy of our location system. Finally, we validate the efficacy of RF hole detection.

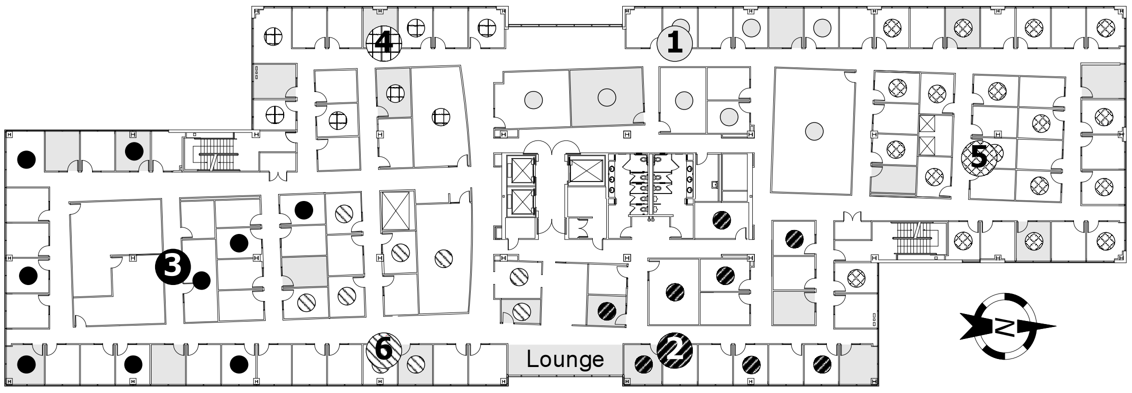


Figure 3: *Floor map. The floor is 98 meters long and 32 meters wide. Numbered circles denote the locations of the six corporate APs. Smaller circles mark the center of rooms containing our AirMonitors. The pattern of an AirMonitor circle's shading matches that of the AP it is configured to track. The shaded rooms are the 21 locations where a client was temporarily placed during the validation experiment.*

5.1.1 Frame Capture Validation

To validate our strategy for capturing frames, we need to show that we can successfully capture most of the traffic between clients located on our floor and the corporate APs that they use. In particular, we must show that at least one AirMonitor is able to hear most of the data packets sent in the uplink direction, and at least one AirMonitor is able to hear most of the data packets sent in the downlink direction. These AirMonitors need not be the same, since the Client Performance Tracker selects the best AirMonitor separately for uplink and downlink traffic. In our validation experiments, we focus on data packets, because management packets (e.g. beacons, probe requests/response) and control packets (ACKs, RTS, CTS) are generally smaller than data packets, and are sent at lower data rates compared to the data packets. Therefore, it is generally easier to overhear these packets.

To carry out the validation, we began by selecting 21 locations on our floor. We chose one location in the public lounge on our floor, which does not have an AirMonitor in it. We chose ten office locations at random from among the set of offices that *have* an AirMonitor in them, and we chose ten more at random from the set of offices that *did not have* an AirMonitor in them. Our tests were conducted in the evening to allow us to enter offices without disturbing the occupants.

The experiment we performed at each location is as follows. We placed a laptop equipped with an 802.11g card in the location, and we performed a 2 MB TCP upload and a 2 MB TCP download over the corporate wireless network. The other end of the TCP connection was a server attached to the corporate wired network. We observed that the vast majority of data packets sent on the air in both directions were sent at the 54 Mbps rate. Note that in each location, the client associated with whichever AP it deemed best; we made no attempt to control this selection. We collected packet traces on both client and the server, to establish the ground truth: the number of IP packets sent by the client and the server. These packets appear as 802.11 Data frames on the air.

For the duration of the test, all the AirMonitors ran with AP tracking enabled, and they also ran a special debugging filter that logs every 802.11 frame it overhears into a local file. We post-processed the AirMonitor 802.11 frame logs to count the number of data frames overheard by each AirMonitor that belonged to our TCP transfers and that did not have the retry bit set, and selected the AirMonitor with the largest frame count. The reason we focused only on frames that did not have the retry bit set is because it is very difficult to know exactly how many retry frames are sent on the air. It may have been possible for us to instrument the client device driver to count the number of retry frames sent, but it was not possible for us to instrument the APs to count retries. On the other hand, using packet traces on the client and server machines allows us to precisely count the number of frames that are sent without the retry bit set.

For each transfer, we calculate the percentage of missed packets by: comparing the frame count from the best uplink AirMonitor with the packet count from the client packet trace; and comparing the frame count from the best downlink AirMonitor with the packet count from the server packet trace. The results of these experiments are shown in Table 2 for all 21 locations. This data shows that the highest percentage of missed packets for any uplink transfer was only 7%, and the highest percentage of missed packets for any downlink transfer was only 3.5%. Previous work has shown [26] that one needs only an 80% capture rate to fully reconstruct the traffic pattern.

Given that our system experiences such low packet loss rate even for short bursts of very high data-rate traffic, we believe that in normal operation, we lose very little traffic from the clients that are located on our floor. The robustness of our deployment, due to the density of AirMoni-

Office	Uplink		Downlink	
	Frames missed (%)	# of AMs that saw > 90% frames	Frames missed (%)	# of AMs that saw > 90% frames
1	2.7	4	1.4	4
2	0.8	1	0.6	5
3	5.6	1	0.6	4
4	3.3	3	3.5	4
5	3.8	4	2.2	4
6	1.3	3	2.6	3
7	1.8	3	2.2	3
8	3.9	4	0.7	14
9	6.4	8	0.3	16
10	1.9	1	1.4	5
11	2.0	3	1.9	5
12	1.5	5	0.8	4
13	1.8	1	1.4	3
14	3.8	2	1.4	5
15	1.8	3	1.3	2
16	4.6	2	3.1	4
17	1.5	4	1.4	3
18	7.0	3	3.4	9
19	5.9	3	2.3	14
20	1.8	3	0.4	4
Lounge	5.2	2	2.0	4

Table 2: *Baseline Data Frame Loss Results. The first 10 offices have AirMonitors in them, the next 10 offices do not. The lounge does not have an AirMonitor.*

Algorithm	Offices With AM			Offices Without AM		
	Min	Med	Max	Min	Med	Max
Strongest AM	0	0	6.5	2.9	3.5	10.4
Centroid	0	0	3.2	0.2	2.5	5.9
Spring and ball	0	0	3	0.9	1.9	6.1

Table 3: *Comparison of three location algorithms. Numbers report error in meters.*

tors, is evident from the fact that for most locations multiple AirMonitors can see more than 90% of the uplink and downlink frames. There were 4 locations where only 1 AirMonitor captured more than 90% of the frames sent in uplink direction. All these locations are at the south end of our floor, where the AirMonitor density is less than at the north end.

5.1.2 Accuracy of the Location Engine

In this section, we evaluate the accuracy in our deployment of the three simple location estimation algorithms described in Section 3.1. To perform this validation, we use the same data collected for the frame capture validation experiment. Along with the debugging filter that logs every 802.11 frame, the AirMonitors also ran another filter that kept track of the average RSSI of the packets sent by the client during the experiment. At the end of the experiment the AirMonitors submitted these average RSSI values to the database. Note that for each client location, only a subset of the AirMonitors hear packets sent by the client. The number of AirMonitors that hear at least one packet depends on a number of factors including the location of the client and which AP the client chooses to associate with.

Note that for this experiment, the AirMonitors record the RSSI of any packet they can attribute as sent by the client. For example, before starting the data transfer, the client may perform an active scan to select an AP by sending probe requests on various channels. If an AirMonitor overhears any of these probe requests, it will include them in its RSSI calculations.

We submit these RSSI values to the location engine to produce an estimate of the client location using each of the three local algorithms. For the spring-and-ball algorithm, we need to use a profile. This profile was generated on a normal weekday between 2 to 3 PM, three days before the location experiment was conducted.

To calculate the location error, we need to calculate the distance between the estimated location and the actual client location, which means that we need to know the actual client location. When we conducted the experiment, we usually placed the laptop on the occupant's office desk. Because there is no easy way for us to determine this actual location, we simply assume that the client was placed at the center of each office. We call this the "assumed actual" (AA) location of the client. This assumption is consistent with the assumption we made during the bootstrapping process when we automatically determined the AirMonitor locations. Because our target for the location system is only office-level accuracy, this assumption is adequate for our purposes.

Table 3 lists the summary of the results of this experiment. Our criteria for "office-level" accuracy is less than 3 meters, since our offices are roughly 10 ft by 10 ft. We see that the StrongestAM algorithm works well only for those offices which contain an AirMonitor, which is as expected. When there is no AirMonitor at the location, the strongest AM is usually in one of the neighboring offices, and we will estimate the client location to be in the center of that office. As a result, the minimum error will be roughly equal to the size of one office, which is what we see from these results. In a few cases, we even see that the StrongestAM algorithm reports a non-zero error when there is an AM in the same location as the client – this happens when an AM in another room reports a higher signal strength.

We also see that the Centroid algorithm performs quite well, and provides office-level accuracy in most cases. We also see that the performance of the Centroid and Spring-and-ball algorithms is generally similar, as one would expect from our deployment. The Centroid algorithm will only generate a large error if the AirMonitors that hear packets from the client at high signal strength are all on one side of the actual client location, and this rarely happens in our deployment, for a couple of reasons. First, clients tend to associate with APs that are near their location, and because each AirMonitor tracks the AP it is closest to, clients are generally surrounded by AirMonitors that are all listening on the same channel the client is transmitting on. In addition, most clients tend to routinely scan on all channels, sending probe requests to discover nearby APs. These probes are overheard by different groups of AirMonitors tracking different APs, and this helps solve the boundary problem. Given the generally good performance of the Centroid algorithm, the Spring-and-ball algorithm can not provide significant improvement in our environment.

We can, however, show specific examples of the boundary problem in our environment. When our laptop client was in the public lounge, we estimated its location using the Centroid algorithm and found the error was 3.6 meters. The Spring-and-ball algorithm, however, has the advantage of the profile information. It correctly realizes that the client is far away from all the AirMonitors that can hear it, and moves the client to minimize the error. As a result, the error reported by the Spring-and-ball algorithm is only 0.6 meters in this case.

We have evaluated the Spring-and-ball algorithm extensively under various conditions, and we found that, in our environment, the improvement it offers over the Centroid algorithm is small. Because the Spring-and-ball algorithm requires generation of profiles, we use the Centroid algorithm for estimating location for the rest of this paper.

While the above validation experiments were carried out at night, we have also extensively validated our system at various other times of the day. We found that the accuracy of the location system is unaffected by factors such as the presence of people in their offices and other traffic on the WLAN.

5.1.3 Validation of RF Hole Detection

The Connectivity Tracker is designed to detect areas of poor RF coverage, from which clients persistently fail to connect to the corporate WLAN. We validated our algorithm and implementation in two ways. First, we removed the antenna of a wireless card so that its transmission range was severely restricted. This card was unable to communicate with the AP, but because of the high density of AirMonitors, we were able to hear its packets, and the Connectivity inference engine located and flagged the client as in an RF hole.

For our second experiment, we set up a test AP in one office, and used a laptop to test the connectivity to the test AP from 12 locations on our floor, as shown in Figure 4. From 4 of these locations, the laptop had good connectivity; from 2 of these locations the Connectivity Tracker flagged a high number of AP switches, indicating unstable connectivity; and from 6 of these locations the Connectivity Tracker detected an RF hole and successfully located each of them. This Figure shows the results of the Connectivity Tracker's classification and location estimate for each test location.

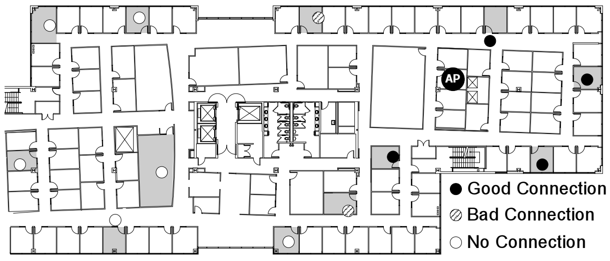


Figure 4: *RF hole validation experiment. We attempted to connect to the shown AP from the shaded offices. The dots show the estimated client locations, and the classification of each location.*

Finally, we also checked for RF holes on the operational WLAN in our building. The Connectivity Tracker did not find any clients that appeared to have difficultly connecting for more than 3 minutes, indicating that all areas of our floor have at least some WLAN coverage.

5.2 Deployment on a Operational Network

In this section, we demonstrate the utility of our system by using it to monitor the corporate WLAN on our floor. We present results generated by both the Client Performance Tracker and the Connectivity Tracker. Unless mentioned otherwise, the results described in this section are calculated over a period of one work-week; Monday, October 2nd through Friday, October 6th, 2006. We only report activity between 8am and 8pm each day, since there is very little traffic on the network outside of these hours. There was one 23 minute period of down-time on October 3rd due to a database misconfiguration.

Because our building has four floors and our testbed is only deployed on one of them, we need to filter out clients who are located on other floors of our building. This is because we cannot report accurate locations for those clients, and because we cannot overhear any of their high data-rate transmissions. We performed experiments where we carried a laptop to the floors above and below our own, and we devised the following threshold based on those experiments: we consider a client to be located on our floor during a particular 15 minute interval if at least four AirMonitors overhear that client, and at least one of them reports a mean RSSI for transmissions from that client of at least -75 dBm.

Figures 5 and 6 provide a high-level view of the amount of activity on our WLAN. Figure 5 shows both the average and the maximum number of active clients on a per-AP basis for the six APs on our floor. The averages are calculated over 15 minute intervals, and "active" simply means that the client exchanged at least one frame with the AP. The number of clients associated with an AP can be as high as 12, although the average is lower. Figure 6

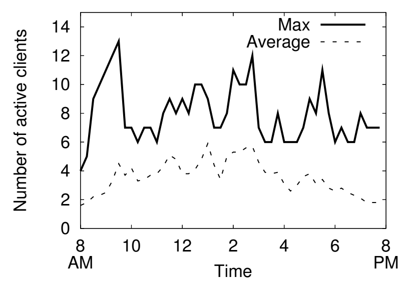


Figure 5: *Number of active clients per AP (15 minute interval)*

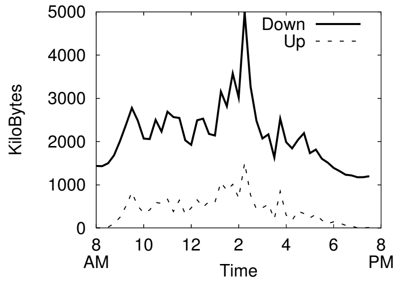


Figure 6: *UpLink and DownLink traffic per AP (15 minute interval)*

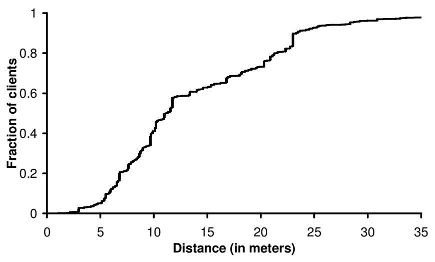


Figure 7: *Distance between clients and the APs they attempt to associate with.*

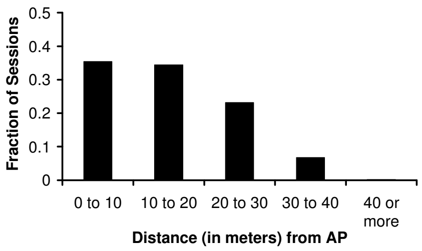


Figure 8: *Distance between APs and clients, when the session lasts longer than 60 seconds.*

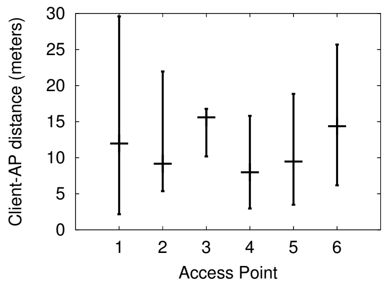


Figure 9: *Distance between active clients and APs: per-AP statistics* (10^{th} *percentile, median and* 90^{th} *percentile).*

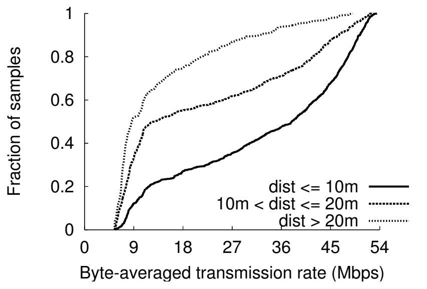


Figure 10: *Impact of distance between clients and APs on transmission rate of downlink traffic. (G clients only).*

shows the average uplink and downlink traffic (in kilobytes), measured over 15 minute intervals. As expected, downlink traffic volume is substantially higher than the uplink traffic volume.

Next, we look at client association patterns. Figure 7 shows the CDF of the distance between clients and the APs they *attempt* to associate with. Not all these association attempts are successful. However, it is interesting to see 40% of the clients attempt to associate with an AP that is over 20 meters away. This is surprising because the average distance to the nearest AP from any location on our floor is only 10.4 meters. While some of this discrepancy may be due to RF propagation effects, we believe that poor AP selection policies implemented in certain Wi-Fi drivers also play a role. While clients do attempt to associate with far-away APs, they generally do not stay associated with them for long. In Figure 8 we look at the connections that lasted for more than 60 seconds. We see that when a client is further than 20 meters from an AP, the probability that it will have a long-lived connection with that AP is small.

We also use client association behavior to evaluate the effectiveness of the AP placement on our floor. In Figure 9, we take a closer look at client distance from the perspective of the 6 APs on our floor. We find that for AP 3, the median distance of associated clients is larger than that of the other APs, and 90% of the clients that associate with this AP are more than 15 meters away. This is due to poor AP placement. This AP is supposed to serve the south end of our building, yet due to the presence of a large lab facility the offices at the far end of the building are too far away from this AP (and they are even further away from all other APs).

We also study the impact of client location on connection quality. We use two metrics to evaluate connection quality: byte-averaged transmission rate, and packet loss rate. For this evaluation, we only consider those clients that transmit at least 50 Data frames on either the uplink or the downlink.

Figures 10 and 11 show the impact of the distance between clients and APs on both uplink and downlink transmission rates. For these two graphs we only considered clients that were operating in 802.11g mode. The filtering is necessary because the maximum transmission rate for 802.11b is only 11 Mbps. These graphs illustrate two interesting points. First, we see that the distance between clients and APs has a significant impact on both uplink and downlink transmission rates. When a client is more than 20 meters away from an AP, the median data rate on both the uplink and the downlink is less than 10 Mbps. We also see that for a given distance, the uplink rates are generally higher than the downlink rates. We believe that this is due to two reasons. First, we measured that the APs on our floor use lower transmission power levels than most client Wi-Fi cards, and this is likely done to minimize interference. Second, the chosen rate is heavily in-

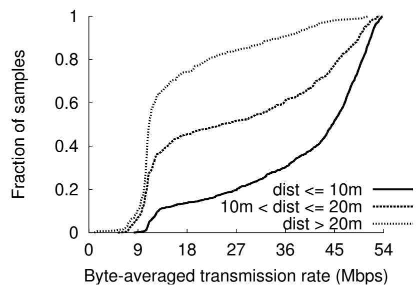


Figure 11: *Impact of the distance between clients and APs on transmission rate of uplink traffic. (G Clients only).*

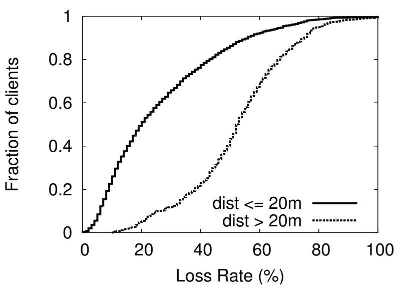


Figure 12: *Impact of the distance between clients and APs on the loss rate of downlink traffic.*

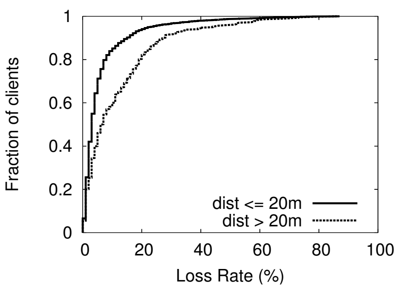


Figure 13: *Impact of the distance between clients and APs on the loss rate of uplink traffic.*

fluenced by the frame loss rate, and as we will see shortly, the downlink loss rates are noticeably higher than the uplink ones.

We also looked at similar graphs for clients that appear to operate only in 802.11b mode. The impact of distance on the transmission rate is much less pronounced for these clients. This is because the maximum transmission rate is capped at 11 Mbps, and the lowest data rates (1 and 2 Mbps) are rarely used even at moderate distances.

Next, we look at the impact of distance on frame loss rates. Figures 12 and 13 show the downlink and uplink loss rates, respectively. We see that downlink loss rates are significantly higher than uplink loss rates. The median downlink loss rate for clients within 20 meters of the AP is 20%, while the median uplink loss rate is only 3%. Mahajan et. al. have reported comparable results in [26]. Furthermore, distance seems to have a much bigger impact on downlink loss rates than it has on uplink loss rates. The median downlink loss rate for clients that are more than 20 meters away from APs is almost 50%. Note that this is the MAC-layer frame loss rate. A 50% MAC layer loss rate, with 4 retransmissions which is typical for most 802.11 devices, would translate into a 6.25% loss rate at the network layer if losses were independent.

The high downlink loss rate may be the result of aggressive rate adaptation algorithms on the AP. As mentioned earlier, our APs transmit at lower power levels than most clients. An aggressive rate adaptation algorithm can cause a high frame loss rate at low transmit power. We investigated in detail those cases when the reported downlink loss rate was higher than 80%. In some of these cases none of the AirMonitors heard any of the ACK packets sent by the client, resulting in incorrect loss estimation, and the total number of frames transmitted was small. These cases account for less than 2% of the samples.

We also considered whether there are any regions on our floor where clients see consistently poor performance. We divided the area of our floor into 14 by 16 meter rectangles. We looked at the median loss rate and the median transmission rate of clients who were located in each regions. One area stood out as worse than the others, and is highlighted in Figure 14. The median downlink loss rate for clients in this area was 49%, which is substantially higher compared to clients in other regions. In addition, the median byte-averaged downlink transmission rates for clients in this area is only 7.3 Mbps. We also noticed that many clients in this area tend to associate with multiple APs, some on other floors. The circles in Figure 14 indicate locations from which the clients connected to at least 5 access points. In most other regions, the clients tend to associate with one or two APs. This indicates that poor AP placement is one potential cause of clients' poor performance in this area.

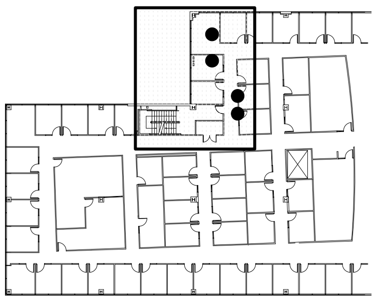


Figure 14: *Region of poor performance. The circles indicate locations where clients connected to at least 5 APs.*

5.3 Effects of Density

We investigate the effects of AirMonitor density on our results by randomly selecting subsets of our 59 AirMonitors and eliminating the data collected by those AirMonitors from our analysis. We then recalculate the graphs shown in the previous section for three different densities: 25%, 50%, and 75%. Reducing the number of AirMonitors reduces the amount of traffic observed, and also reduces the accuracy of the location results. At 25% density, the total amount of traffic observed during the entire one week period is reduced to 78% of the original total, and at 50% density it is reduced to 95% of the original. We believe that many of the frames that are lost in the lower density

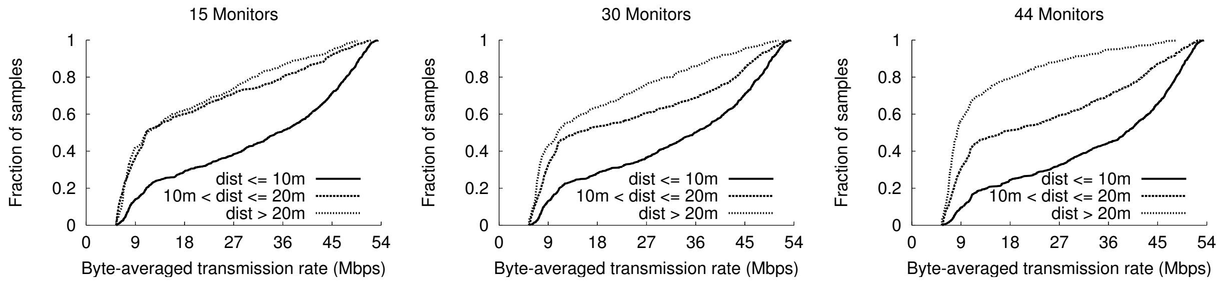


Figure 15: *Each figure shows the impact of the distance between clients and APs on downlink transmission rates. The three figures are calculated using AirMonitor densities of 25%, 50%, and 75%, respectively.*

configurations are those frames sent at higher transmission rates, and we are currently investigating this further.

For many of the performance statistics, we see qualitative differences in the results at densities of 25% and 50%, whereas the 75% density results tend to look quite similar to the 100% results. For example, in Figure 15, the three side-by-side graphs show the relationship between downlink transmission rates and the distance between clients and the APs to which they are associated. The leftmost graph shows the results for 25% AirMonitor density, the middle graph shows the results for 50% density, and rightmost graph shows the 75% results. For comparison, the 100% density results are shown in the previous section in Figure 10.

6 Discussion

In this section, we discuss various issues related to the performance and effectiveness of our system.

• Our system determines the locations of the AirMonitors by guessing the primary user of the machine, and then reading the building map. We repeat this process periodically to ensure accuracy of the AirMonitor locations.

If a machine is moved without its location being updated, the client locations that relied on observations from the moved AirMonitor may be in error. So far, we have not had any problems with unannounced machine moves. We note that we can automatically detect when a machine is moved far away from its original location. Each AirMonitor is aware of its neighbors, and if the neighbor set changes significantly, the system administrator can be notified. However, if the AirMonitor is moved only a short distance (say to the next office), we may not be able to automatically detect such a move. In such cases, the error introduced in the estimated client locations is also likely to be small.

• We have not studied the accuracy of our location system in settings other than our office floor. It is likely that the accuracy of the location system, and specifically, that of the Centroid algorithm, is helped by the fact that our offices have full-height walls and solid wood doors. The radio signal fades rapidly in such an environment ensuring that only nearby AirMonitors hear a strong signal from a given transmitter. In other settings (e.g. cubicles), many more AirMonitors may receive a strong signal from a given transmitter. This may degrade the accuracy of the Centroid algorithm.

• Our system currently requires that each AirMonitor track the channel of nearest the AP. This strategy is appropriate for the management applications described in this paper, as clients in the region near an AP are also likely to associate that AP, and therefore operate on the same channel as our AirMonitors. Other channel assignment strategies may be appropriate for other applications. For example, for wireless security applications, it is desirable to monitor as much of the spectrum as possible in each area. This can be accomplished by having each AirMonitor randomly cycle through all channels, spending some time on each. We are also considering signal-strength based channel assignment.

• Because scalability is an important design goal, we briefly summarize measurements of our system's scalability. During the full work week of operation, the size of all our database tables grew by just under 10 GB. Each AirMonitor generated less than 11 Kbps of traffic on the wired network submitting summaries to the database. We monitored the additional CPU load we placed on the AirMonitors, and this load rarely exceeds 3%.

• Both the loss rate and the transmission rate statistics calculated by the Client Performance Tracker are *estimates*. For transmission rates, AirMonitors are less likely to overhear high transmission rate frames, so it is likely that we are underestimating the true rates. We may overestimate loss rates if no AirMonitor overhears many of the Data packets sent without the retry bit set. However, the results of our validation experiment in Table 2 give us confidence that this is unlikely.

7 Related work

Location Estimation: Many researchers have built systems for location estimation in WiFi networks. RADAR [7] uses pattern matching of received signal strength at a client from various landmarks, such as APs, to locate a Wi-Fi client. Further refinements of the idea appear in several subsequent papers [23, 22, 30, 10, 17]. These schemes include a manual profiling phase, which requires the network operator to collect RF fingerprints from a large number of locations. Many of these schemes are designed to allow WiFi clients to locate themselves - not for the system to determine where the client is. We note that some of the techniques described in these papers can be used to enhance the accuracy of our system as well.

ActiveCampus [9] uses a hill-climbing algorithm to approximate the location of a mobile user using samples from multiple APs. The accuracy can be further enhanced by using feedback from mobile users.

The Placelab project [24] incorporates a self-mapping system that requires knowledge of only a few anchor nodes. This system is geared towards outdoor environments and the average location error is 31 meters.

Location estimation has also been studied for wireless sensor networks. Systems like MoteTrack [25] are designed for emergency management environments, and require extensive prior manual profiling. Sextant [16] uses a small number of landmark nodes to determine the region of a sensor node. It works by formulating geographic constraints on the location of the node. A key idea behind Sextant is to use information about which nodes *can not* hear each other. In our system, we can not be certain about the transmit power of the clients. Therefore we can not directly use Sextant's model.

Location systems that use non-RF technologies such as ultrasound [29] are outside the scope of this work.

Research wireless network monitoring systems: Several researchers have studied mobility and traffic patterns in enterprise wireless networks. In [8], the authors characterize wireless traffic and user mobility in a large corporate environment using SNMP traces. They can not directly observe either user location or phenomena such as MAC-layer retransmissions. In [21, 18], the authors study traffic and usage patterns in an academic wireless network. In [20], the authors study usage patterns of WLAN at at IETF meeting. In these studies, the user location is characterized by the AP that the users are associated with. Our results show that users sometimes associate with APs that are far away.

Jigsaw [12] is a WLAN monitoring system that uses multiple monitors and trace combining to generate a comprehensive view of network events. There are two key differences between Jigsaw and our system. First, Jigsaw does not include a location engine and hence when it is used to detect performance problems it cannot provide specific information about the location(s) of the client(s) experiencing those problems. Second, Jigsaw is architecturally is very different from our system. Jigsaw uses dedicated, custom-built, multi-radio monitoring nodes whereas we add off-the-shelf USB wireless adapters to end-user desktops. Jigsaw generates full packet traces at every monitoring node and performs a centralized merge of these traces, whereas our monitors generate application-specific traffic summaries, and we attempt to select the "best" monitor for hearing a particular communication to avoid having to merge observations from multiple monitors. With full packet traces, Jigsaw can provide a detailed look at low-level network effects such as interference, that DAIR can only guess at.

In [15], the authors describe a simple packet capture utility and how they used it for intrusion detection. WIT [26] is a toolkit for analyzing the MAC-level behavior of WLANs using traces captured from multiple vantage points. The toolkit does not include a location estimation tool. We have discussed the relationship between WIT and our approach throughout the paper.

In [4], the authors present a system for diagnosing faults in Wi-Fi networks. They study some of the same problems that we do, and their techniques for locating disconnected clients are somewhat similar to ours. However, their system is based entirely on observations made by mobile clients. It requires installation of special software on clients. Furthermore, the accuracy of their location system is significantly worse than ours: they report a median error of 9.7 meters.

Commercial wireless network monitoring systems: Currently, a few commercial systems are available for wireless network monitoring and diagnosis. Many of these systems focus on security issues such as detecting rogue APs, which was the subject of our previous paper [5].

Unlike research systems, only a few selective details are publicly available about the performance of commercial systems. Below, we present a summary comparison of DAIR with two commercial monitoring systems, using the claims they make in their marketing literature and white papers available on their websites.

AirTight Networks [2] sells a system called SpectraGuard, which has many features similar to DAIR. Spectraguard, however, requires the use of special sensor nodes that have to be manually placed in various places within the area to be monitored. It also requires special software to be installed on the clients. The system is focused on detecting threats such as rogue access points. Spectraguard also provides RF coverage maps and some performance diagnostics, although the details are not available in the published data sheets. Spectraguard includes a location subsystem. However, details about how it works, and what accuracy it provides are unavailable.

AirDefense [1] also sells a system for monitoring of corporate wireless networks. Like SpectraGuard, this product also focuses on wireless security problems. It also requires use of special wireless sensors, and installation of special software on end systems. The key differentiating feature of this product is called "RF Rewind". The sensors collect upto 270 measurements per wireless device per minute. This data can be later used for forensic investigations. The product includes some location determination features, although the details are not publicly available. It is not clear what performance diagnostic information is provided by AirDefense.

8 Conclusion

We have built a scalable, and easy-to-deploy WLAN management system that incorporates a self-configuring location engine. Using our system, we monitored an operational WLAN network. Our results show the importance of knowing the locations of WLAN clients in understanding their performance. Therefore, we have demonstrated the need for a location estimation engine to be an integral part of any WLAN management system. We have also demonstrated that by using a dense deployment of wireless sensors, one can use simple algorithms to estimate client location and performance. Finally, we showed that a dense deployment of wireless monitors can be achieved in a cost-effective manner using the DAIR platform.

Although the focus of this paper has been on using location to improve the performance and reliability of WLANs, the location engine we have built is also directly applicable to the wireless security management applications that were the focus of our previous DAIR paper [5].

Acknowledgments

We would like to thank Victor Bahl, Ratul Mahajan, John Dunagan, Krishna Gummadi and the anonymous reviewers for their helpful comments.

References

[1] AirDefense: Wireless LAN Security. http://airdefense.net.

[2] AirTight Netwoks. http://airtightnetworks.net.

[3] Aruba Networks. http://www.arubanetworks.com.

[4] A. Adya, P. Bahl, R. Chandra, and L. Qiu. Architecture and techniques for diagnosing faults in IEEE 802.11 infrastructure networks. In *MOBICOM*, 2004.

[5] P. Bahl, R. Chandra, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. Enhancing the security of corporate wi-fi networks using DAIR. In *MobiSys*, 2006.

[6] P. Bahl, J. Padhye, L. Ravindranath, M. Singh, A. Wolman, and B. Zill. DAIR: A framework for managing enterprise wireless networks using desktop infrastructure. In *HotNets*, 2005.

[7] P. Bahl and V. N. Padmanabhan. RADAR: An in-building rf-based user location and tracking system. In *INFOCOM*, 2000.

[8] M. Balazinska and P. Castro. Characterizing mobility and network usage in a corporate wireless local-area network. In *MobiSys*, 2003.

[9] E. Bhasker, S. W. Brown, and W. G. Griswold. Employing user feedback for fast, accurate, low-maintenance geolocationing. In *IEEE Percom*, 2004.

[10] P. Castro, P. Chiu, T. Kremenek, and R. Muntz. A probabilistic room location service for wireless networked environments. In *UBICOMP*, 2001.

[11] R. Chandra, J. Padhye, A. Wolman, and B.Zill. A Location-Based Management System for Enterprise Wireless LANs. Microsoft Research Technical Report MSR-TR-2007-16. February 2007.

[12] Y.-C. Cheng, J. Bellardo, P. Benko, A. Snoeren, G. Voelker, and S. Savage. Jigsaw: Solving the puzzle of enterprise 802.11 analysis. In *SIGCOMM*, 2006.

[13] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A decentralized network coordinate system. In *SIGCOMM*, 2004.

[14] J. R. Douceur and W. J. Bolosky. Progress-based regulation of low-importance processes. In *SOSP*, 1999.

[15] S. Felis, J. Quittek, and L. Eggert. Measurement based wireless LAN troubleshooting. In *WinMee*, 2005.

[16] S. Guha, R. Murty, and E. G. Sirer. Sextant: A unified node and event localization framework using non-convex constraints. In *MobiHoc*, 2005.

[17] A. Haeberlen, E. Flannery, A. M. Ladd, A. Rudys, D. S. Wallach, and L. E. Kavraki. Practical robust localization over large-scale 802.11 wireless networks. In *MOBICOM*, 2004.

[18] T. Henderson, D. Kotz, and I. Abyzov. The changing usage of a mature campus-wide wireless network. In *MOBICOM*, 2004.

[19] IEEE802.11b/D3.0. Wireless LAN Medium Access Control(MAC) and Physical (PHY) Layer Specification: High Speed Physical Layer Extensions in the 2.4 GHz Band, 1999.

[20] A. Jardosh, K. Ramachandran, K. C. Almeroth, and E. M. Belding-Royer. Understanding congestion in IEEE 802.11b wireless networks. In *IMC*, 2005.

[21] D. Kotz and K. Essien. Analysis of a campus-wide wireless network. In *MOBICOM*, 2002.

[22] J. Krumm and E. Horvitz. Locadio: Inferring motion and location from WiFi signal strengths. In *Mobiquitous*, 2004.

[23] A. M. Ladd, K. E. Bekris, A. Rudys, G. Marceau, L. E. Kavraki, and D. S. Wallach. Robotics-based location sensing using wireless ethernet. In *MOBICOM*, 2002.

[24] A. LaMarca, J. Hightower, I. Smith, and S. Consolvo. Self-mapping in 802.11 location systems. In *Ubicomp*, 2005.

[25] K. Lorincz and M. Welsh. Motetrack: A robust, decentralized approach to RF-based location tracking. In *LoCA*, 2005.

[26] R. Mahajan, M. Rodrig, D. Wetherall, and J. Zahorjan. Analyzing MAC-level behavior of wireless networks in the wild. In *SIGCOMM*, 2006.

[27] Microsoft Active Directory. http://www.microsoft.com/windows/server2003/technologies/directory/activedirectory/.

[28] D. Mills. Network time protocol (version 3) specification, implementation and analysis. RFC 1305, March 1992.

[29] N. B. Priyantha, A. Miu, and H. Balakrishnan. The Cricket location-support system. In *MOBICOM*, 2001.

[30] M. A. Youssef, A. Agrawala, and A. U. Shankar. WLAN location determination via clustering and probability distributions. In *IEEE Percom*, 2003.

Beyond One-third Faulty Replicas in Byzantine Fault Tolerant Systems

Jinyuan Li* and David Mazières
VMware Inc. and Stanford University

ABSTRACT

Byzantine fault tolerant systems behave correctly when no more than f out of $3f + 1$ replicas fail. When there are more than f failures, traditional BFT protocols make no guarantees whatsoever. Malicious replicas can make clients accept arbitrary results, and the system behavior is totally unspecified. However, there is a large spectrum between complete correctness and arbitrary failure that traditional BFT systems ignore. This paper argues that we can and should bound the system behavior beyond f failures.

We present BFT2F, an extension to the well-known Castro-Liskov PBFT algorithm [6], to explore the design space beyond f failures. Specifically, BFT2F has the same liveness and consistency guarantees as PBFT when no more than f replicas fail; with more than f but no more than $2f$ failures, BFT2F prohibits malicious servers from making up operations that clients have never issued and restricts malicious servers to only certain kinds of consistency violations. Evaluations of a prototype implementation show that the additional guarantees of BFT2F come at the cost of only a slight performance degradation compared to PBFT.

1 INTRODUCTION

Applications with high security needs can reduce the danger of attacks through Byzantine-fault-tolerant (BFT) replication. A service replicated over several BFT servers can survive even when some fraction of the replicas fail (become malicious). Thus, failures whose likelihood is not correlated across machines have a much smaller probability of affecting overall application security. In particular, if replicas are separately administered, BFT replication protects against malicious or incompetent machine operators. To a limited extent, BFT replication also protects against software errors in components with different implementations on different machines.

BFT protocols generally aspire to two properties [6]: *linearizability* [9] and *liveness*. Linearizability, also called *ideal consistency* or *safety*, is the property that the service appears to all clients to execute an identical sequence of requests, and that this sequence preserves the temporal order of non-concurrent operations. Liveness means that the system makes progress in executing clients' requests, at least under some weak assumptions about eventual message delivery.

Unfortunately, all BFT protocols make a strong assumption that some predetermined fraction of server replicas are honest. In particular, the highest fraction of failures that an asynchronous BFT system can survive without jeopardizing linearizability or liveness is f out of $3f + 1$ replicas. The reason is that asynchronous communication makes it impossible to differentiate slow replicas from failed ones. To progress safely with f unresponsive replicas, a majority of the remaining $2f + 1$ responsive ones must be honest.

The security of today's best-known BFT algorithms fails completely given even just $f + 1$ compromised replicas. For example, an attacker who compromises two out of four servers can return arbitrary results to any request by any client, including inventing past operations that were never requested by any user, rolling back history to undo operations that were already revealed to clients, or producing illegal results that could not have arisen from any valid sequence of operations. However, depending on the situation, it may be harder for an attacker to compromise two thirds of replicas than one third. A system that guarantees some residual security properties when a third or more of the replicas fail could significantly improve security in such settings.

Fortunately, linearizability and total failure are not the only options. The goal of this work is to limit the damage when more than f out of $3f + 1$ servers in a replicated state system fail. Specifically, we explore a new, weaker consistency model, fork* consistency, a derivative of fork consistency introduced earlier [16]. With fork* consistency, it is possible to bound a system's behavior when between $f+1$ and $2f$ replicas have failed. When $2f+1$ or more replicas fail, it is unfortunately not possible to make any guarantees without simultaneously

*Research done while the author was visiting Stanford University.

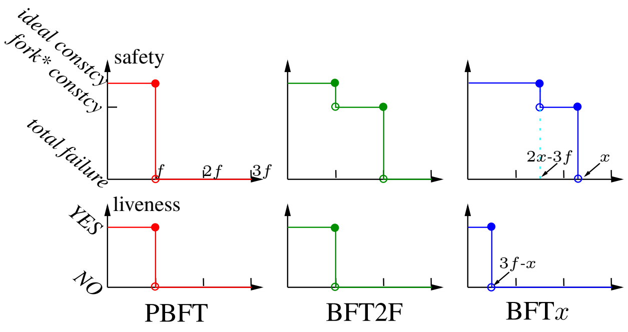


Figure 1: Comparison of the safety and liveness guarantees of PBFT, BFT2F, and BFTx. As we can see, BFT2F provides extra safety guarantees without compromising liveness, which is strictly better than PBFT.

sacrificing liveness for cases where f or fewer replicas fail.

We propose BFT2F, a protocol that provides exactly the same linearizability guarantee as PBFT [6] when up to f replicas have failed. When more than f but no more than $2f$ replicas have failed, two outcomes are possible. (1) The system may cease to make progress—In other words, BFT2F does not guarantee liveness when more than f replicas are compromised. Fortunately, for most applications, denial of service is less harmful than arbitrary behavior. (2) The system may continue to operate and offer fork* consistency, which again for many applications is preferable to arbitrary behavior.

Fork* consistency is perhaps best motivated by example. Consider a campus on which people swipe ID cards to open locked doors. Suppose the server that processes access requests is replicated on four machines. With PBFT, an attacker who compromises two replicas can open any door on campus—even doors he has never had access to—without leaving an audit trail. With fork* consistency, the attacker might be able to delay revocation requests and gain access to rooms he should no longer have access to. He may also complicate recovery by causing his accesses to be logged on only one correct replica instead of all correct replicas. However, he will not gain access to new rooms or be able to unlock doors without leaving some evidence behind.

In practice, surviving up to $2f$ malicious replicas enables replicated state systems for the first time to handle the case where a *majority* of the replicas may fail. Meanwhile, BFT2F does not compromise any guarantees that existing BFT systems provide. One might wonder if there exist protocols that guarantee fork* consistency with even larger numbers of failures, e.g., x failures for $2f < x \leq 3f$. The answer is yes. In fact, BFT2F can be easily parameterized to support that, which we term BFTx in Figure 1. However, achieving fork* consistency with more than $2f$ failures comes at the cost of sacrificing liveness when there are fewer than f failures. Administrators who care more about security than availability can tune the protocol to guarantee fork* consistency anywhere up to $3f$ failures. In fact, through a degenerate case, a system configured to guarantee fork* consistency with as few as one correct replica actually provides linearizability, at the cost of sacrificing liveness in the face of even one failure.

The paper is organized as follows. We start by specifying the details of the system model and our assumptions in Section 2. Section 3 gives a formal specification of fork consistency and proves that no protocols can guarantee fork consistency using only one round of communication when more than f servers fail. In Section 4, we relax the model of fork consistency further to arrive at fork* consistency, which is achievable using a one round protocol and still provides useful guarantees. We present BFT2F, an extension to the classical PBFT protocol to achieve fork* consistency when more than f servers are faulty in Section 5. We evaluate a prototype implementation of BFT2F in Section 6. Lastly, we discuss related work in Section 7 and conclude in Section 8.

2 Models and Assumptions

We use the same replicated state machine model as presented in past work [6, 24]. The network is unreliable and may replicate, discard, and reorder messages with the one exception that it cannot indefinitely prevent two honest parties from communicating with each other.

The system consists of a number of clients and $3f+1$ replicated servers, where f is a predetermined parameter. Clients make requests for operations to servers. An operation is *executed* when servers accept it and apply it to their state. An operation is *completed* when its originating client accepts the reply. We use lower-case letters that appear early in the alphabet such as a, b, c to denote clients, and letters that appear late in the alphabet such as p, q, u, w to denote servers. We also use the term replicas and servers interchangeably. A node is either a client or server. Each node has a public/private key pair. Each server knows the public keys of all nodes and each client knows the public keys of all servers.

We assume a Byzantine failure model. A faulty node may either halt or deviate from correct behavior arbitrarily, even colluding with other faulty nodes. However, a faulty node cannot forge digital signatures of correct nodes or find collisions of a cryptographic hash function.

3 Fork consistency

Figure 2 shows the server-side interface between a state-machine replication system (such as PBFT) and the replicated service (e.g., an NFS file server). The service starts in a fully-specified initial state and provides a single function, EXECUTE, that takes a request operation, *op*, as an argument, then deterministically updates its state and returns a result, *res*. In ordinary operation, clients submit requests through a client-side replication library. The server-side code communicates amongst replicas so as to invoke EXECUTE with an identical sequence of requests on every replica, thereby producing identical results to send back to the client. The client side library returns the results from the matching replies to the client.

Of course, a compromised server need not behave as described above. It can fail to respond to messages. Worse yet, it may return arbitrary results to clients; it may subvert network routing to intercept and tamper with packets; it may even collude with other compromised servers and send carefully-crafted messages that subvert honest replicas and ultimately convince the client library to return incorrect or inconsistent results. We need techniques that allow honest clients and replicas to minimize the damage that may be caused by malicious replicas.

A significant part of the challenge is ensuring that malicious replicas do not corrupt the internal state of honest replicas. Because an honest replica's internal state is entirely determined by the sequence of all operations it has ever executed, we introduce a term for this history: We call the sequence $L^{all} = \langle op^1, op^2, ..., op^n \rangle$ of all operations executed by a replica the *result list* of its latest operation op^n. This result list entirely determines both the result returned by the EXECUTE function for op^n and the state S^{n+1} of the replica after executing op^n. Whenever possible, we would like to ensure that honest replicas all have the same result lists and that these lists contain only requests legitimately issued by users.

To defend against illegitimate requests, each operation may be signed by the key of its originating user. Without knowing a user's private key, a bad replica cannot forge requests. By verifying the signatures on requests, honest replicas can limit themselves to calling EXECUTE only on operations legitimately requested by users. Should malicious replicas learn a compromised user's private key, they may be able to convince honest replicas to execute arbitrary operations with the privileges of the compromised user; effectively any well-formed operation by a compromised user is legitimate.

In an ideally consistent (well-behaved) system, every honest replica will have the same result list $L^{all} = \langle op^1, op^2, ..., op^n \rangle$ for operation op^n. Moreover, this list will preserve the temporal order of non-overlapping operations. Malicious replicas, of course, may behave as though op^n's result is *not* the outcome of executing L^{all}, but rather of some different result list L^{bad}. In fact, malicious server behavior may be incompatible with *any* possible sequence of legitimate requests; we therefore allow L^{bad} to contain illegal operations (such as "cause the next operation to return this particular result").

```
// At server side:
($S^{n+1}$, $res$) ← EXECUTE($S^n$, $op$);
```

Figure 2: Pseudocode for replicated state machines.

When a system undergoes total failure (as in the case of PBFT with more than f faulty replicas), an operation's result list even at an honest replica might not contain all previously issued requests; malicious servers may have succeeded in concealing the existence of legitimately issued operations. Worse yet, it may be that different replica's result lists contain the same operations in different orders, and that this convinces the client libraries on different machines to return inconsistent results. To mitigate these problems, we previously defined a consistency model, fork consistency [16], that can be realized in the presence of entirely malicious servers. An important property of fork consistency is that it helps provide server accountability by allowing clients to detect any past consistency violations on the part of bad servers.

In defining fork consistency, we make a distinction between operations issued by *correct* clients, which obey the protocol, and *malicious* or compromised clients that don't. We use the term result lists only for replies to operations issued by correct clients running our library, as it makes little sense to talk about the results seen by clients that do not implement the protocol or even necessarily use our software. However, we consider any result accepted by a correct client to have a result list. In the event that malicious servers completely break the system, they may induce clients to accept "impossible" results, in which case the corresponding result lists must have illegal operations.

Definition: A system is *fork consistent* if and only if it satisfies the following requirements:

- **Legitimate-request Property:** Every result accepted by a correct client has a result list L that contains only well-formed operations legitimately issued by clients.

- **Self-consistency Property:** Each honest client sees all past operations from itself: if the same client a

issues an operation op^i before op^j, then op^i also appears in op^j's result list.

This property ensures that a client has a consistent view with respect to its own operations. For example, in a file system, a client always reads the effect of its own writes.

- **No-join Property:** Every accepted result list that contains an operation *op* by an honest client is identical up to *op*.

One corollary of fork consistency is that when two clients see each others' recent operations, they have also seen *all* of each other's past operations in the same order. This makes it possible to audit the system; if we can collect all clients' current knowledge and check that all can see each others' latest operations, we can be sure that the system has *never* in the past violated ideal consistency. Another corollary is that every result list preserves the temporal order of non-concurrent operations by correct clients, since an accepted result list cannot contain operations that have not yet been issued.

We say two result lists are *consistent* if one is an improper prefix of the other. We say two result lists are *forked* if they are not consistent. For instance, result list $\langle op^1, op^2\rangle$ and result list $\langle op^1, op^3\rangle$ are forked, while $\langle op^1, op^2\rangle$ and $\langle op^1, op^2, op^3\rangle$ are not. If clients accept forked result lists, the system has failed to deliver ideal consistency.

We say servers are in different *fork sets* if they reflect forked result lists. A fork set consists of a set of servers who return the same set of consistent result lists to all clients, though in an asynchronous system, some results may not reach the client before an operation completes. Intuitively, a correct server cannot be in more than one fork set. A malicious server, however, can simultaneously be in multiple fork sets—pretending to be in different system states when talking to different nodes. An attacker who controls the network and enough replicas may cause the remaining honest replicas to enter different forks sets from one another, at which point with any practical protocol clients can no longer be assured of ideal consistency.[1]

Figure 3 shows an example where fork consistency differs from ideal consistency. Initially, all clients have the same result list $\langle op^1\rangle$. Client a issues a request op^2, and gets back the result list $\langle op^1, op^2\rangle$ from fork set

[1]Technically, a contrived protocol could permit honest replicas to enter forked states that clients could always detect. However, such a protocol would be needlessly vulnerable; since honest replicas know the system's complete state, they can and should detect any attacks clients can, rather than purposefully entering divergent states that would be more difficult to recover from.

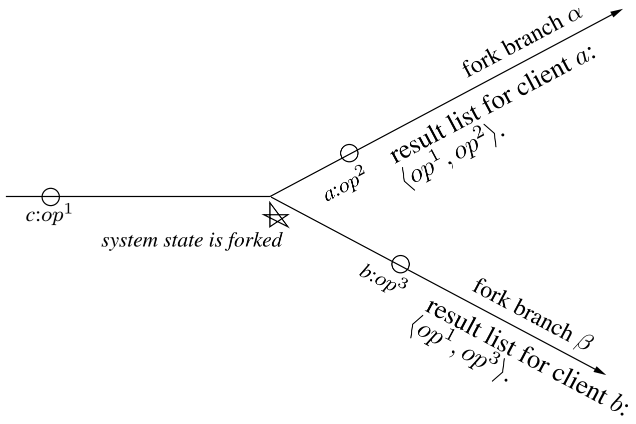

fork branch α
result list for client a: $\langle op^1, op^2\rangle$.
a:op^2
c:op^1
system state is forked
b:op^3
fork branch β
result list for client b: $\langle op^1, op^3\rangle$.

Figure 3: An example of fork consistency. Since the server deceives client b about a's operation op^2, client a's result list $\langle op^1, op^2\rangle$, and b's result list $\langle op^1, op^3\rangle$ are only fork consistent, not ideally consistent. (Strictly speaking, client a still has ideal consistency at this moment, should op^2 be ordered before op^3. However, client a will miss ideal consistency hereafter.)

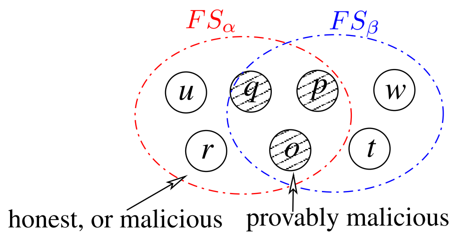


Figure 4: In a replicated state system, the intersection of two fork sets can only consist of *provably malicious* servers, which are denoted by shaded circles. The partition that excludes the intersection part might have honest servers and malicious ones that have not yet misbehaved.

FS_α; after that, client b's operation op^3 returns with the result list $\langle op^1, op^3\rangle$ from fork set FS_β. Here the server deceives b about a's completed operation op^2. Therefore, client a's result list and client b's are forked. At this moment, one can verify that the system has delivered fork consistency, but failed to provide ideal consistency. It is worth pointing out that if a protocol returned the entire result list with each reply to an operation, a client could compare result lists to detect the attack whenever it received replies from two different fork sets. The BFT2F protocol presented in Section 5 uses hashes to achieve a similar guarantee without the efficiency or privacy problems of shipping around result lists.

Figure 4 shows a potential configuration that could produce the two fork sets in Figure 3. $FS_\alpha \bigcap FS_\beta$ must have *only* faulty servers that have already acted maliciously (e.g., p, q, o). $FS_\alpha - (FS_\alpha \bigcap FS_\beta)$ and $FS_\beta - (FS_\alpha \bigcap FS_\beta)$ can include either honest servers (e.g., u, w, t), or malicious servers who have not misbehaved so far (e.g., r could be such a server), and so who are externally indistinguishable from honest servers, though they might later deviate from the protocol.

3.1 Fork consistency examples

For some applications, fork consistency may be nearly as good as ideal consistency. In the example of a card-swipe access control service, if two out four replicas are compromised and an attacker forks the system state, it is likely only a matter of hours or days before people notice something is wrong—for instance that new users' cards do not work on all doors. At that point, the administrators must repair the compromised replicas. In addition, they may need to merge access logs and replay all revocations executed in either fork set by the two honest replicas. Even without server compromises, adding and removing door access is often not instantaneous anyway, both because of bureaucracy and because people do not always report lost or stolen ID cards immediately.

On the other hand, in other settings fork consistency may lead to quantifiable damage compared to ideal consistency. Consider a bank that uses four machines to replicate the service that maintains account balances. An attacker who compromises two of the machines and forks the state can effectively duplicate his account balance. If the attacker has $1,000 in his account, he can go to one branch and withdraw $1,000, then go to a branch in a different fork set and withdraw $1,000 again.

Finally, of course, there are settings in which fork consistency on its own is nearly useless. For example, a rail signaling service split into two fork sets, each of which monitors only half the trains and controls half the signals, could easily cause a catastrophic collision. One way to mitigate such problems is to leverage fork consistency to get bounded inconsistency through heartbeats. A trusted heartbeat client could increment a counter every few seconds; trains that do not detect the counter being incremented could sound an alarm. The attacker would then need to compromise either another f replicas or the heartbeat box to suppress the alarms.

3.2 Impossibility of one-round fork consistency

Unfortunately, for a system to guarantee fork consistency when the fraction of honest replicas is too small for linearizability, clients must send at least two messages for every operation they perform. The most serious implication of this is that the system can no longer survive client failures—if a client fails between the first and second messages of an operation, the system may no longer be able to make progress. Another disadvantage is that slow clients can severely limit system throughput by leaving servers idle between the two messages. Finally, there are various implementation reasons why a one-round protocol is often simpler and more efficient, particularly if there is high latency between the client and servers.

Theorem: In an asynchronous system that provides liveness when up to f out of $3f + 1$ replicas fail, guaranteeing fork consistency in the face of more than f replica failures requires more than one round of client-server communication on each operation.

Proof Sketch: For simplicity, we consider the case of four replicas with $f = 1$, though the same argument applies to any number of replicas. In a single-round protocol, a client sends a single message that eventually convinces honest replicas to alter their state by executing an operation. Consider the case when two clients, a and b, issue two requests, op_a and op_b, concurrently. Neither client could have known about the other's request when issuing its operation. Thus, either op_a or op_b is capable of being executed before the other.

If, for instance, the network delays op_a, op_b could execute before op_a arrives and vice versa. Moreover, because of liveness, request op_a must be capable of executing if both client b and replica w are unreachable, as long as the remaining three replicas respond to the protocol. Figure 5 illustrates this case, where the result list $\langle op_b, op_a \rangle$ was eventually returned to client a.

The same reasoning also applies to client b, who might get result list $\langle op_a, op_b \rangle$ when replica u is unreachable. These two out-of-order result lists reflect the (malicious) servers' ability to reorder concurrent requests at will, if a single message from a client is allowed to change the system state *atomically*. This clearly violates the no-join property of fork consistency. ∎

3.3 A two-round protocol

The problem with a one-round protocol is that at the time a client signs a request for operation op, there may be previous operations by other clients it does not know about. Therefore, there is nothing the client can include in its signed request message to prevent the two honest replicas (u and w) from believing op should execute in two different contexts.

In a two-round protocol, the client's first message

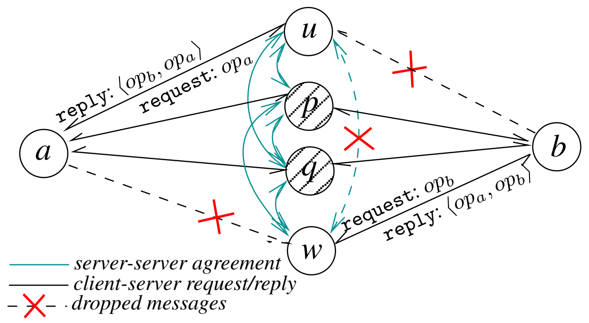


Figure 5: Two malicious servers (p and q) wear different hats when talking to distinct honest servers (u or w) in the server-server agreement phase, during which all servers communicate with each other to achieve consensus on incoming requests. Meanwhile, they delay the communication between u and w. In this way, p and q, with u, return result list $\langle op_b, op_a\rangle$ to client a; p and q, with w, return $\langle op_a, op_b\rangle$ to client b.

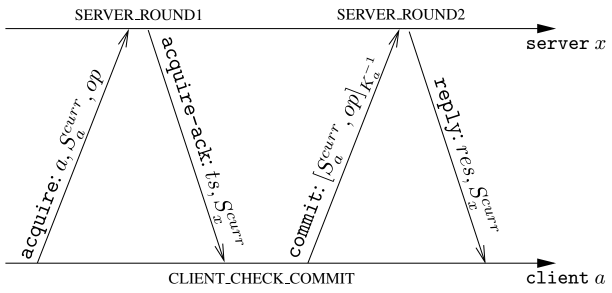

SERVER_ROUND1
SERVER_ROUND2
server x
acquire: a, S_a^{curr}, op
acquire-ack: ts, S_x^{curr}
commit: $[S_a^{curr}, op]_{K_a^{-1}}$
reply: res, S_x^{curr}
CLIENT_CHECK_COMMIT
client a

Figure 6: A two-round protocol.

can request knowledge about any previous operations, while the second message specifies both the operation and its execution order. Figure 6 shows such a protocol. A client sends an `acquire` request to acquire the system's latest state in the first round; a server replies with its current state. Disregarding efficiency, a straw-man protocol might represent this state as the previous operation's result list (i.e., the impractically long list of every operation ever executed). In the second round, the client commits its operation *op* by signing a message that ties *op* to its execution order. Again, in a straw-man, inefficient protocol, the client could append its new operation to the list of all previous operations and sign the whole list. Servers then sanity check the commit message, execute *op*, and send back the reply.

Note that servers are not allowed to abort an operation after replying to an `acquire` message—the `acquire-ack` is a commitment to execute the operation in a particular order given a signed `commit`. Otherwise, if servers could optionally abort operations, malicious replicas p and q could convince u and a that op_a aborted while convincing w to execute it, violating fork consistency. But of course a server cannot execute an operation until it sees the signed `commit` message, which is why clients can affect both liveness and throughput.

These problems can be overcome in application-specific ways. When it is known that an operation's result cannot depend on a previous operation's execution, it is safe to execute the later operation before the previous one commits. SUNDR performs this optimization at the granularity of files and directory entries, which mostly overcomes the performance bottleneck but can still lead to unavailable files after a client failure. Because SUNDR does whole-file overwrites, one can recover from mid-update client crashes by issuing a new write that either deletes or supersedes the contents of whatever file is generating I/O timeout errors.

HQ replication [8] similarly allows concurrent access to different objects. The technique could be used to improve performance of a two-round protocol. However, object semantics are general enough that there would be no way to recover from a client failure and still provide fork consistency with $f+1$ malicious replicas.

Figure 7 shows a two-round protocol in pseudocode. S_n^{curr} represents node n's latest knowledge of the system state. We say $S_i^{curr} \prec S_j^{curr}$ (meaning S_i^{curr} *happens before* S_j^{curr}) if one of node j's past system states is identical to S_i^{curr}—in other words, node j's current knowledge is consistent with and strictly fresher than i's. In our straw-man protocol, if S_n^{curr} is a list of all operations ever executed, then $S_i^{curr} \prec S_j^{curr}$ means S_i^{curr} is a prefix or S_j^{curr}.

4 Fork* consistency

Because of the performance and liveness problems of two-round protocols, this section defines a weaker consistency model, fork* consistency, that can be achieved in a single-round protocol. Intuitively, rather than have separate `acquire` and `commit` messages, our plan will be to have each request specify the precise order in which the same client's *previous* request was supposed to have executed. Thus, it may be possible for an honest replica to execute an operation out of order, but at least any future request from the same client will make the attack evident.

Turning back to the example in Figures 3 and 4, with a one-round protocol, after the system state has been forked, client c's operation op^4 might be applied to servers in both fork sets if c has no clue that the system state has been forked when it issues op^4. Then the honest servers in both FS_α and FS_β will update their system state, S_α or S_β, respectively, with op^4, violating the no-join property of fork consistency. However, client

```
//executed by server $x$ upon receiving acquire.
procedure SERVER_ROUND1($a, S_a^{curr}, op$)
  if $next = null$ and $S_a^{curr} \prec S_x^{curr}$
      Extract timestamp $ts$ from $op$;
      Agree with other servers that current state is $S_x^{curr}$
        and next operation to execute is $op$ from client $a$;
      $next \leftarrow [a, op]$;
      send [acquire-ack, $ts, S_x^{curr}$] to client $a$;


//executed by client $a$ to collect acquire-acks
//and generate commit.
procedure CLIENT_CHECK_COMMIT($op$)
  Decide on the system's current state $S^{curr}$ based on
    $2f + 1$ matching acquire-acks from servers;
  $S_a^{curr} \leftarrow S^{curr}$;
  send $[\texttt{commit}, S_a^{curr}, op]_{K_a^{-1}}$ to all servers;


//executed by server $x$ upon receiving commit.
procedure SERVER_ROUND2($a, S_a^{curr}, op$)
  if $next = [a, op]$ and $S_a^{curr} = S_x^{curr}$
      $(S_x^{curr}, res) \leftarrow$ EXECUTE($S_x^{curr}, op$);
      send [reply, $S_x^{curr}, res$] to client $a$;
      $next \leftarrow null$;
```

Figure 7: Pseudocode for a two-round protocol.

c is only going to accept the reply from one of the fork sets, e.g., FS_α, and adopt its new state, S_α, as its freshest knowledge of the system state. If the protocol can prohibit any correct node that has seen state in one fork set, namely FS_α, from issuing any operations that execute in another (e.g., FS_β), then all future operations from client c can only be applied to servers in fork set FS_α (or subsequent forks of FS_α).

Fork* consistency therefore relaxes the no-join property in fork consistency to a join-at-most-once property: two forked result lists can be joined by at most one operation from the *same* correct client.

- **Join-at-most-once Property:** If two result lists accepted by correct clients contain operations op' and op from the same correct client, then both result lists will have the operations in the same order. If op' precedes op, then both result lists are identical up to op'.

The join-at-most-once property is still useful for system auditing. We can periodically collect all clients' current knowledge to check that all have seen each others' latest operations. Suppose each client issues an operation for this purpose at times t^1, t^2 ... t^n, where t^n is the time of the latest check. The join-at-most-once property guarantees that if the checks succeed, then the system has *never* violated ideal consistency up to time t^{n-1}.

In the example above, for client a, the new result list is $\langle op^1, op^2, op^4\rangle$, and for b, it is $\langle op^1, op^3, op^4\rangle$. Yet, the system still delivers fork* consistency at this moment, just not fork consistency or ideal consistency. However, fork* consistency forces malicious servers to choose between showing c's future operations to a or b, but not both.

4.1 Fork* consistency examples

In the card-swipe example, fork* consistency has effectively the same consequences as fork consistency. It delays revocation and complicates recovery. The only difference is that after the attack one operation from each client may appear in both fork sets, so, for example, one user may get access to a new door in both fork sets, but no one else will.

In the case of the bank, fork* consistency increases the potential loss. For example, the attacker can go to one branch, withdraw \$1,000, then go to the other branch and deposit the \$1,000. The attacker can ensure this deposit shows up in both fork sets—allowing him to withdraw an additional \$1,000 from the first branch, as well as the \$2,000 he can get from the branch at which he made the deposit.

In the case of a heart-beat server used to sound an alarm, fork* consistency doubles the amount of time required to detect a problem, since one increment of the counter may show up in both fork sets.

5 BFT2F ALGORITHM

In this section, we present BFT2F, an extension of the original PBFT protocol [6] that guarantees fork* consistency when more than f but not more than $2f$ out of $3f + 1$ servers fail in an asynchronous system.

Like PBFT, BFT2F has replicas proceed through a succession of numbered configurations called *views*, and uses a three-phase commit protocol [22] to execute operations within views. The three phases are *pre-prepare*, *prepare*, and *commit*. In view number $view$, replica number $view \bmod 3f + 1$ is designated the *primary* and has responsibility for assigning a new sequence number to each client operation.

5.1 BFT2F Variables

Below, we describe major new variables and message fields introduced in BFT2F. We use superscripts to denote sequence number, e.g., msg^n refers to the message with sequence number n. We use subscripts to distinguish variables kept at different nodes.

Hash Chain Digest (HCD): A HCD encodes all the operations a replica has committed and the commit order. A replica updates its current HCD to be $HCD^n = D(D(msg^n) \circ HCD^{n-1})$ upon committing msg^n, where D is a cryptographic hash function and $\circ$ is a concatenation function. Replicas and clients use HCDs to verify if they have the same knowledge of the current system state.

Hash Chain Digest History: To check if a replica's current knowledge of the system state is strictly fresher than another replica's and not forked, each replica keeps a history of past HCDs. We denote replica p's history entry for msg^n as $T_p[n]$.

Version Vector: Every node i represents its knowledge of the system state in a version vector V_i. The version vector has $3f+1$ entries, one for each replica. Each entry has the form $\langle r, view, n, HCD^n \rangle_{K_r^{-1}}$, where r is the replica number, $view$ is the view number, n is the highest sequence number that node i knows replica r has committed, and HCD^n is r's HCD after n operations. The entry is signed by r's private key ${K_r}^{-1}$. We denote replica r's entry in V_i by $V_i[r]$, and use C structure notation to denote fields—i.e., $V_i[r].view$, $V_i[r].n$, and $V_i[r].HCD$.

We define several relations and operations on these data structures:

- Let V be a version vector. We define a *cur* function to represent the current state of the system in V as follows: If at least $2f+1$ entries in V have the same n and HCD values, $\mathrm{cur}(V)$ is one of these entries (e.g., the one with the lowest replica number). Otherwise, $\mathrm{cur}(V) = \mathbf{none}$.
- Let h and g be two version vector entries signed by the same replica. We say h *dominates* g iff $h.view \geq g.view$ and either the two entries have identical n and HCD fields or $h.n > g.n$.
- We say h *dominates* g *with respect to* hash chain digest history T iff h dominates g and the two entries' HCD fields appear at the appropriate places in T—i.e., $h.HCD = T[h.n]$ and $g.HCD = T[g.n]$. In other words, this means that g appears in the history T that leads up to h.

Whenever a client a sees a version vector entry h signed by replica p, it always updates its own version vector V_a by setting $V_a[p] \leftarrow h$ if h dominates the old value of $V_a[p]$. A server r similarly updates $V_r[p] \leftarrow h$ if h is recent (e.g., not older than the beginning of T_r) and dominates the old $V_r[p]$ with respect to T_r.

5.2 BFT2F Node Behavior

In describing BFT2F, we borrow heavily from PBFT [6]. However, we point out two major differences between BFT2F and PBFT. First, unlike in PBFT, BFT2F replicas do not allow out of order commits. This requirement does not pose much overhead as replicas must execute client operations in increasing sequence numbers anyway. Second, BFT2F requires clients to wait for at least $2f+1$ matching replies before considering an operation completed, as opposed to the $f+1$ matching replies required in PBFT.

5.2.1 *Client Request Behavior*

A client a multicasts a request for an operation $\langle \texttt{request}, op, ts, a, \mathrm{cur}(V_a) \rangle_{K_a^{-1}}$ to all the replicas, where ts is a monotonically increasing timestamp, $\mathrm{cur}(V_a)$ is a's last known state of the system, and the message is authenticated by a's signature key, ${K_a}^{-1}$.

We note that while PBFT uses more efficient MAC vectors to authenticate requests from clients, BFT2F requires public-key signatures. The reason is that faulty clients may interfere with the replicas' operation by issuing requests in which some MAC vector entries are valid and some are invalid. Such a partially correct MAC vector causes some replicas to accept the request and some to reject it. PBFT can recover from such a situation if $f+1$ replicas attest to the validity of their MAC vector entries. However, in BFT2F, we want to avoid executing illegitimate operations even with $> f$ faulty replicas, which means it is still not safe to execute an operation under such circumstances.

5.2.2 *Server Behavior*

Every server keeps a replay cache containing the last reply it has sent to each client. Upon receiving a request $\langle \texttt{request}, op, ts, a, \mathrm{cur}(V_a) \rangle_{K_a^{-1}}$ from client a, a server first checks the signature, then checks that the request is not a replay. Clients execute one operation at a time, so if ts is older than the last operation, the server ignores the request. If ts matches the last operation, the server just re-sends its last reply. Otherwise, the server checks that $\mathrm{cur}(V_a).HCD$ matches the HCD in the last reply the server sent to the client. If it does not, the client may be in a different fork set, or may be malicious and colluding with a malicious server. Either way, the server ignores this request. Once a message has been validated and checked against the replay cache, processing continues differently on the primary and other servers.

The primary replica, p, assigns the request a sequence number n and multicasts a `pre-prepare` message $\langle \texttt{pre-prepare}, p, view, n, D(msg^n) \rangle_{\sigma_p}$ to all other replicas. Here σ_p is either a MAC vector or a signature with p's private key, ${K_p}^{-1}$.

Upon receiving a `pre-prepare` message matching an accepted request, replica q first checks that it has not accepted the same sequence number n for a different message ${msg'}^n$ in the same view $view$. It also ensures n is not too far out of sequence to prevent a malicious primary from exhausting the sequence number space. Replica q then multicasts a `prepare` message $\langle \texttt{prepare}, q, view, n, D(msg^n)\rangle_{\sigma_q}$ to all other replicas.

A replica u tries to collect $2f$ matching `prepare` messages (including one from itself) with the same sequence number n as that in the original `pre-prepare` message. When it succeeds, we say replica u has *prepared* the request message msg^n. Unlike PBFT, u does not commit out of order, i.e., it enters the *commit* phase only after having prepared the message *and* committed all requests with lower sequence numbers.

To start committing, replica u computes $HCD^n \leftarrow D(msg^n \circ HCD^{n-1})$ and sets $T_u[n] \leftarrow HCD^n$, updates $V_u[u]$ to $\langle u, view, n, HCD^n\rangle_{{K_u}^{-1}}$, and multicasts a `commit` message $\langle \texttt{commit}, \langle u, view, n, HCD^n\rangle_{{K_u}^{-1}}\rangle$ to all other replicas.

When replica w receives a `commit` message from u with a valid signature and HCD^n, it updates the entry for u in its current version vector, $V_w[u]$, to $\langle u, view, n, HCD^n\rangle_{{K_u}^{-1}}$. Replica w commits msg^n when it receives $2f+1$ matching `commit` messages (usually including its own) for the same sequence number n and the same HCD (HCD^n).

Replica w executes the operation after it has committed the corresponding request message msg^n. It sends a reply message to the client containing the result of the computation as well as its current version vector entry $V_w[w]$. Since w has collected $2f+1$ matching `commit` messages, we know that these $2f+1$ replicas are in the same fork set up to sequence number n.

5.2.3 Behavior of Client Receiving Replies

A reply from replica w has the format, $\langle \texttt{reply}, a, ts, res, \langle w, view, n, HCD^n\rangle_{{K_w}^{-1}}\rangle_{\sigma_w}$, where $view$ is the current view number, ts is the original request's timestamp, and res is the result of executing the requested operation. A client considers an operation completed after accepting at least $2f+1$ matching replies each of which contains the same ts, res, n, and HCD^n. (Recall by comparison that PBFT only requires $f+1$ matching replies.) This check ensures the client only accepts a system state agreed upon by at least $2f+1$ replicas. Therefore, if no more than $2f$ replicas fail, the accepted system state reflects that of at least one correct replica. Client a also updates its $V_a[w]$ to $\langle w, view, n, HCD^n\rangle_{{K_w}^{-1}}$ for each w of the $2f+1$ replies, ensuring that HCD^n becomes the new value of $\text{cur}(V_a).HCD$.

To deal with unreliable communication, client a starts a timer after issuing a request and retransmits if it does not receive the required $2f+1$ replies before the timer expires. Replicas discard any duplicate messages and can also fetch missing requests from each other in case the client crashes.

5.3 Garbage Collection

If a replica r has been partitioned from the network, it may have missed some number of successfully executed operations and need to learn them from other replicas. For small numbers of missed operations, the replica can just download the logged operations and `commits` and execute any operation that has $2f+1$ `commit` messages with appropriate HCDs. However, if other replicas have truncated their logs, r may not be able to download all missing operations individually. It may instead need to do a bulk transfer of the entire state of the service from other replicas. The question then becomes how to authenticate the state in such a transfer.

In PBFT, r validates the state using stable checkpoints gathered every I operations (e.g., $I = 128$). A stable checkpoint is a collection of signed messages from $2f+1$ replicas attesting that the service had hash $D(\text{state}^n)$ at sequence n. r can then believe state^n. In BFT2F, the signed messages must additionally vouch that state^n is in the same fork set as r and allow r to bring its replay cache up to date. Our implementation accomplishes this by having each replica keep state back to n_{low}, the lowest version number in its version vector. This state may be required for application-specific recovery from fork attacks anyway. However, it requires unbounded storage while any replica is unavailable, which may be undesirable, so here we outline a way to achieve fork* consistency with bounded storage.

When replica u signs a checkpoint for sequence n, it includes its version vector V_u in the message. If no w has $V_u[w].n \leq n-2I$, then no honest replica will ever ask to be updated from a state $n-2I$ or older. If, however, there is a $V_u[w] \leq n-2I$, then, for each such w, u includes in the checkpoint one of its own old version vector entries h that dominates $V_u[w]$ with respect to T_u. Furthermore, it keeps enough commit messages around to roll w's state forward from $V_u[w].HCD$ to $h.HCD$, so that w can be assured it is in the same fork set as h. To ensure w does not progress beyond $h.n$, replicas execute no more than $2I$ operations beyond the last stable checkpoint that shows them up to date.

In detail, u's signed checkpoint has the form $\langle \texttt{checkpoint}, r, n, D(\text{state}^n), D(\text{rcache}^n), V_u, E\rangle_{K_u^{-1}}$. Here rcache^n is u's replay cache at sequence n (without the signatures or replica numbers, to make it identical at all unforked replicas). V is u's current version vector. E is a set of older values of $V_u[u]$ computed as follows. For each w in which $V_u[w].n \leq n - 2I$, E contains u's signed version vector entry from the next multiple of I after $V_u[w].n + I$. A stable checkpoint consists of `checkpoint` messages signed by $2f + 1$ different replicas with the same n, $D(\text{state}^n)$, $D(\text{rcache}^n)$, and E (modulo the replica IDs and signatures in E). Given a stable checkpoint, u can delete log state except for operations more recent than $n - 2I$ and the $2I$ operations leading up to each element of E.

When computing a stable checkpoint, it may be that replicas do not initially agree on E. However, each replica updates its version vector using any recent signed entry with a valid HCD in any other replica's `checkpoint` message and multicasts a new `checkpoint` message upon doing so. (To ensure the receiving replica can check the HCD, replicas ignore `commit` and `checkpoint` messages before the last stable checkpoint, so that once marked stale in a stable checkpoint, a replica can only change to being completely up-to-date.) Note that even after a stable checkpoint exists, a replica that was previously shown as out of date can cause a second stable checkpoint to be generated for the same sequence number by multicasting a `checkpoint` message with its up-to-date version vector.

5.4 Server View Change

In BFT2F, a replica r experiencing a timeout sends a `view-change` message $\langle \texttt{view-change}, \mathit{view} + 1, V_r[r], P\rangle_{K_r^{-1}}$ to all other replicas. Here $V_r[r]$ is the version vector entry for r's last committed operation, while P is a set of sets P_m for each prepared message m with sequence number higher than n. Each P_m contains the `pre-prepare` message for m and $2f$ corresponding matching `prepare` messages.

The primary p in the new view $\mathit{view} + 1$ checks all signatures (but not MACs) and validates the $V_r[r].\mathit{HCD}$ value in `view-change` messages it receives. If $V_p[p]$ dominates $V_r[r]$ with respect to T_p, then $V_r[r]$ is valid. Otherwise, if $V_r[r]$ dominates $V_p[p]$, then p requests from r an operation and $2f + 1$ matching `commits` with appropriate HCDs for every sequence number between $V_p[p].n$ and $V_r[r].n$. p then executes these operations, bringing T_p up to date so that $V_p[p]$ dominates $V_r[r]$ with respect to T_p. If p cannot download the missing operations from r, it does a bulk state transfer.

We say two `view-change` messages *conflict* if their P fields include different operations for the same sequence number. This may happen if honest replicas are forked because of more than f failures, or if the σ authenticators on `pre-prepare` and `prepare` messages have corrupted MAC vectors and malicious replicas claim to have prepared messages for which they didn't actually receive a `pre-prepare` and $2f + 1$ matching `prepares`. As long as there are $2f + 1$ honest replicas (without which we cannot guarantee liveness), we will be in the latter case and p will eventually receive $2f + 1$ valid and non-conflicting `view-change` messages (including one from itself), at which point it multicasts a `new-view` message $\langle \texttt{new-view}, \mathit{view} + 1, \mathcal{V}, O\rangle_{K_p^{-1}}$. Here $\mathcal{V}$ is the set of $2f + 1$ valid, non-conflicting `view-change` messages. O is a set of `pre-prepare` messages constructed as below:

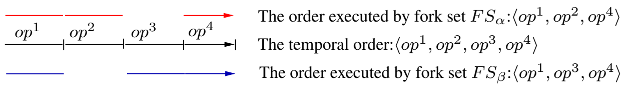
1. p determines *min-s* as the lowest sequence number of any version vector entry ($V_r[r]$) in an element of $\mathcal{V}$. p then determines *max-s* as the highest sequence number of any of the P_m sets in elements of $\mathcal{V}$.

2. For each sequence number n from *min-s* through *max-s*, p either (1) constructs a `pre-prepare` message in the new view, if a P_m in one of the view-change messages has a valid request for sequence number n, or (2) constructs a special *null* request $\langle\langle \texttt{pre-prepare}, p, \mathit{view} + 1, n, D(\mathit{null})\rangle_{K_p^{-1}}, \mathit{null}\rangle$ to fill in the sequence number gap.

A backup replica u in the new view validates the `new-view` message as follows. u checks the version vector in each element of $\mathcal{V}$ using the same checks p performed upon receiving the `view-change` messages. If u is too far out of date, it may need to do a state transfer. u also verifies that O is properly constructed by executing the same procedure as the primary. If u accepts the `new-view` message as valid, it sends a `prepare` message for each message in O and proceeds normally in the new view.

When there are no more than f faulty replicas, the above algorithm is essentially the same as PBFT with hash chain digests replacing the state hashes in PBFT's checkpoint messages. When more than f but no more than $2f$ replicas fail, there may be concurrent view changes in different fork sets. In the worst case when $2f$ replicas fail, up to $f + 1$ view changes may succeed concurrently, leading to $f + 1$ fork sets.

However, with no more than $2f$ failures, each fork set is guaranteed to contain at least one honest replica, r, which ensures fork* consistency. To see this, we con-



op^1
op^2
op^3
op^4
The order executed by fork set FS_α:$\langle op^1, op^2, op^4\rangle$
The temporal order:$\langle op^1, op^2, op^3, op^4\rangle$
The order executed by fork set FS_β:$\langle op^1, op^3, op^4\rangle$

Figure 8: Example of two forked result lists. The middle timeline shows the result list that should have been executed by a non-faulty system. The top timeline shows a forked result list that does not reflect op^3, while the bottom timeline shows another one missing operation op^2.

sider two cases. First, suppose r does not do a state transfer. Its hash chain digests are then simply the result of processing a sequence of operations in turn. Because r checks the $\mathrm{cur}(V_a)$ in a client's request against the last reply to that client, r will never accept a request from an honest client that has executed an operation in another fork set. At worst, r may execute the first request a client executes in a different fork set—hence achieving the join at most once property.

On the other hand, if r does a state transfer, this requires $2f + 1$ other replicas to vouch for r's new state. At least one of those $2f + 1$ replicas—call it u—must also be honest and in r's fork set. Since u is honest, it ensures that any operations r skips executing because of the state transfer do not violate fork* consistency.

5.5 An Example

We demonstrate the join-at-most-once property of BFT2F during normal case operation using a simple example. As in the example from Section 3, the system consists of four replicas u, p, q, w, with p being the primary in the current view and the two replicas p, q being malicious.

First we explain the intuition for why the join-at-most-once property can be achieved with a one-round protocol. Suppose the system is forked into two fork sets between two operations op^1 and op^4 issued successively by client c. Then c's second operation, op^4, might show up in two fork sets because the HCD c includes in the request for op^4 is taken from the reply to op^1, on which all replicas agreed. However, the replies c accepts for op^4 can only come from one or the other of the fork sets. Any subsequent operation c issues will only be valid in that particular fork set.

Now we consider this example in detail: client c issues the first (op^1) and fourth operation (op^4), and some other clients issue the second (op^2) and third (op^3) operations. The result list $\langle op^1, op^2, op^3, op^4\rangle$ would have reflected the order assigned in an otherwise non-faulty system, as shown in Figure 8. The malicious primary, p, assigns sequence number 1 to c's first operation op^1 and shows it to all other replicas. Subsequently, p only shows the second operation, op^2, to u and the third operation, op^3, to w, but it assigns both op^2 and op^3 the same sequence number 2. As a result, two fork sets FS_α and FS_β are formed, where FS_α contains u which has seen $\langle op^1, op^2\rangle$ and FS_β contains w which has seen $\langle op^1, op^3\rangle$.

Replica p then manages to join the two forked result lists for the first time with the operation op^4; the two result lists become $\langle op^1, op^2, op^4\rangle$ and $\langle op^1, op^3, op^4\rangle$, respectively. Suppose client c gets the required $2f+1 = 3$ replies for op^4 from fork set $FS_\alpha = \{u, p, q\}$. Then c's version vector will contain $HCD^3 = D(op^4 \circ D(op^2 \circ D(op^1)))$, while replica w in FS_β has a different version vector V_w containing $HCD'^3 = D(op^4 \circ D(op^3 \circ D(op^1)))$ (shown in Figure 9 Part (i)). Hereafter, if malicious servers p and q try to join the two forked result lists with a another operation by c, say op^5, the HCD^3 included in c's request would conflict with that in w's replay cache (Figure 9 Part (ii)).

5.6 Discussion

Ideal consistency requires *quorum intersection*; any two quorums should intersect in at least one non-faulty replica. Fork* consistency requires only *quorum inclusion*; any quorum should include at least one non-faulty replica. BFT2F uses quorums of size $2f+1$. However, as depicted in Figure 1, this can be generalized to a parameterized BFTx protocol that ensures quorum inclusion up to x failures with quorums of size $x+1$. With $3f+1$ replicas, any two quorums of size $x + 1$ must overlap in at least $2(x + 1) - (3f + 1) = 2x - 3f + 1$ replicas, which guarantees quorum intersection up to $2x - 3f$ failures. Unfortunately, *quorum availability*, or liveness, requires that there actually be $x + 1$ replicas willing to participate in the protocol, which is only guaranteed up to $3f + 1 - (x + 1) = 3f - x$ failures.

BFT2F sets $x = 2f$. When up to f replicas fail, this provides quorum intersection. Because any two quorums share a non-faulty replica, the protocol can guarantee that any operation executed in one quorum is reflected in every subsequent quorum, just as in PBFT. When more than f but no more than $2f$ replicas fail, the quorums become fork sets. Quorum inclusion ensures at least one honest replica in each fork set. The honest replica in turn ensures that the quorum only executes legitimate operations and enforces join at most once, but it cannot guarantee ideal consistency or liveness.

BFT2F aims for full compatibility with PBFT and, subject to PBFT's liveness guarantees, an optimal guarantee beyond f replica failures. Another potentially useful construction is BFT3F, with $x = 3f$. BFT3F requires each quorum to be of size $3f + 1$—i.e., to contain

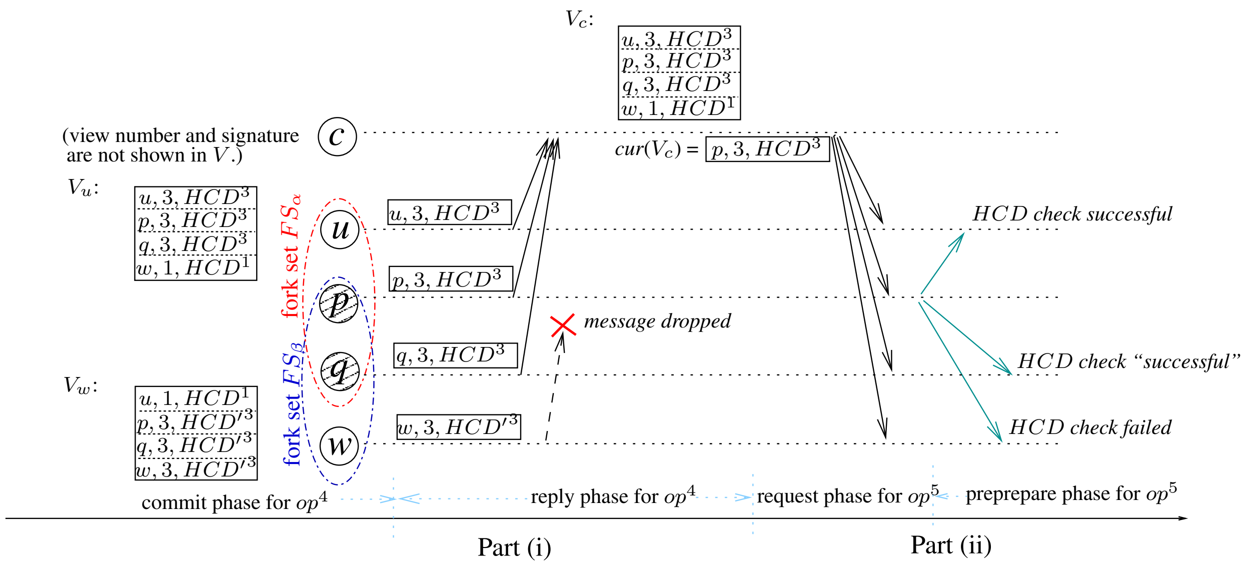


Figure 9: Example of the join-at-most-once property. op^4 is used to join two forked result lists as in Figure 8. Part (i) shows op^4's commit and reply phases. Since the result lists have already been forked, honest replicas in the two fork sets have different HCD histories by sequence number 3—$T_u[3] = HCD^3$ for u and $T_w[3] = HCD'^3$ for w—and thus include different HCDs in their replies. Client c accepts the reply from fork set FS_α and updates V_c accordingly. (c might detect a Byzantine failure upon seeing w's divergent reply in FS_β, but here the network drops the message.) Part (ii) shows that no future operation from c can execute in FS_β, since HCD^3 won't match w's last reply to c.

every replica in the system. This has two implications. First, since only *one* fork set can be formed, ideal consistency will be achieved with up to $3f$ failures. Second, the system will lose liveness with even one unresponsive replica, because no quorum can be formed under such circumstances. A system may want to use BFT3F if safety is the highest priority.

Both SUNDR [11] and BFT2F use version vectors to represent a node's knowledge of the current system state, with one major difference. SUNDR's version vectors have one entry per client, while BFT2F's use one per server. This difference brings BFT2F two advantages. First, BFT2F's version vectors are smaller, since there are typically fewer replicas than clients. Second, in both protocols, a node does not see any updates for version vector entries of nodes in a different fork set. In SUNDR, a stagnant entry could innocently signify that a client is simply offline or idle, both legitimate states. In contrast, BFT2F's replica servers should always remain online to process operations, so that a stagnant version vector entry is a good indication of failure.

6 Performance

We built a prototype implementation of the BFT2F algorithm on FreeBSD 4.11, based on our ported version of the BFT [6] library.

6.1 Implementation

BFT2F's additional guarantee over BFT [6] to allow detection of past consistency violations comes at the expense of much increased use of computationally expensive public key signatures instead of the symmetric session-key based Message Authentication Codes (MACs) used by PBFT. We use NTT's ESIGN with key length of 2048 bits in BFT2F. On a 3 GHz P4, it takes 150 μs to generate signatures, 100 μs to verify. For comparison, 1280-bit Rabin signatures take 3.1 ms to generate, 27 μs to verify.

All experiments run on four machines; three 3.0 GHz P4 machines and a 2.0 GHz Xeon machine. Clients run on a different set of 2.4 GHz Athlon machines. All machines are equipped with 1-3 GB memory and connected via a 100 Mbps Ethernet switch. Performance results are reported as the average of three runs. In all cases, standard deviation is less than 3 % of the average value.

6.2 Micro benchmark

Our micro benchmark is the built-in *simple* program in BFT, which sends a *null* operation to servers and waits for the reply. Appending version vector entries in `request` and `reply` messages has the most impact to the slowdown of BFT2F, compared to BFT.

req/rep(KB)	BFT	BFT(ro)	BFT2F	BFT2F(ro)
0/0	1.027	0.200	2.240	0.676
0/4	1.029	0.778	2.242	1.600
4/0	4.398	3.486	5.647	3.942

Table 1: Latency comparison of BFT and BFT2F (in milliseconds).

6.3 Application-level benchmark

We modify NFS to run over BFT2F, and compare it to the native BSD NFSv2, NFS-BFT running on 4 servers, and SUNDR (NVRAM mode) running on 1 server. The evaluation takes five phases: (1) copy a software distribution package `nano-1.2.5.tar.gz` into the file system, (2) uncompress it in place, (3) untar the package, (4) compile the package, (5) clean the build objects.

	NFSv2	NFS-BFT	NFS-BFT2F	SUNDR
P1	0.302	0.916	1.062	0.299
P2	1.161	3.546	4.131	0.520
P3	2.815	4.171	5.666	1.668
P4	3.937	4.296	4.922	3.875
P5	0.101	0.778	1.707	0.361
Total	8.316	13.707	17.488	6.723

Table 2: Performance comparison of different file system implementations (in seconds).

As Table 2 shows, the application-level performance slowdown in NFS-BFT2F relative to NFS-BFT is much less than that observed in our micro benchmark. This is because the high cost of public key operations and transferring version vector entries accounts for a smaller fraction of the cost to process requests. Both BFT2F and NFS-BFT achieve much lower performance than NFSv2 and SUNDR, reflecting the cost of replication.

7 Related work

Byzantine fault tolerant systems generally fall into two categories: replicated state machines [21] and Byzantine quorum systems [12, 13, 14, 25]. PBFT and BFT2F build on replicated state machines. By contrast, Quorums have simpler construction and are generally more scalable [1]. However, quorums usually provide only low-level semantics, such as read and write, which makes building arbitrary applications more challenging. Quorums also exhibit poor performance under contention. Replicated state machines generally deal with contention more efficiently, but scale poorly to larger numbers of replicas. Many other BFT systems [18, 10, 5] take this approach. Some wide area file systems [19, 2] run BFT on their core servers. HQ replication [8] unifies the quorum and state machine approaches by operating in a replicated state machine mode during high contention and in quorum mode during low contention.

Some work has been done to harden BFT systems against the probability of more than f simultaneous failures. Proactive recovery [7] weakens the assumption of no more than f faulty replicas during the lifetime of the service to no more than f failures during a window of vulnerability. It achieves this by periodically rebooting replicas to an uncompromised state. However, it's behavior is still completely undefined when more than f replicas fail in a given window. Furthermore, some problems such as software bugs persist across reboots. BASE [20] aims to reduce correlated failures. It abstracts well-specified state out of complex systems, and thus reduces the chances of correlated software bugs by allowing the use of different existing mature implementations of the same service.

By separating execution replicas from agreement replicas [24], one can tolerate more failures within execution replicas or reduce replication cost. BAR [3] takes advantage of the fact that selfish nodes do not fail in completely arbitrary ways. Dynamic Byzantine quorum systems [4] can adjust the number of replicas to tolerate varying f on the fly, based on the observation of system behavior.

Securing event history has been studied in the systems like timeline entanglement [15], which takes the hash chain approach as in BFT2F, and in [17, 23], which use version vectors to reason about partial ordering.

8 Conclusion

Traditional BFT algorithms exhibit completely arbitrary behavior when more than f out of $3f+1$ servers are compromised. A more graceful degradation could improve security in many settings. We propose a weak consistency model, fork* consistency, for BFT systems with larger numbers of failures. Fork* consistency prevents clients from seeing the effects of illegitimate operations and allows detection of past consistency violations. We present a new algorithm, BFT2F, based on PBFT, that provides the same guarantees as PBFT when no more than f replicas fail, but offers fork* consistency with up to $2f$ faulty replicas. While BFT2F does not guarantee liveness in the latter situation, denial of service is far preferable to arbitrary behavior for most applications. Evaluation of our prototype BFT2F implementation suggest its additional guarantees come with only a modest performance penalty.

ACKNOWLEDGMENTS

We thank Michael Freedman, Jinyang Li, Antonio Nicolosi, Junfeng Yang, Jinyuan's thesis committee, the anonymous reviewers, and our shepherd Barbara Liskov, for comments on the paper, as well as the National Science Foundation for support under grants CCR-0093361 and CNS-0430425.

REFERENCES

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 59–74, Brighton, United Kingdom, October 2005. ACM.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, available, and reliable storage for an incompletely trusted environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 1–14, December 2002.

[3] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles*, pages 45–58, Brighton, United Kingdom, October 2005. ACM.

[4] Lorenzo Alvisi, Dahlia Malkhi, Evelyn Pierce, Michael K. Reiter, and Rebecca N. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the the International Conference on Dependable Systems and Networks (FTCS 30 and DCCA 8)*, pages 283–292, June 2000.

[5] Christian Cachin and Jonathan A. Poritz. Secure intrusion-tolerant replication on the internet. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 167–176, 2002.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 173–186, New Orleans, LA, February 1999.

[7] Miguel Castro and Barbara Liskov. Proactive recovery in a Byzantine-fault-tolerant system. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 273–288, San Diego, CA, October 2000.

[8] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pages 177–190, Seattle, WA, November 2006.

[9] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems*, 12(3):463–492, 1990.

[10] Kim Potter Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. The SecureRing group communication system. *ACM Transactions on Information and System Security*, 4(4):371–406, 2001.

[11] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 121–136, December 2004.

[12] Dahlia Malkhi and Michael Reiter. Byzantine quorum system. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 569–578, El Paso, TX, May 1997.

[13] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the 7th IEEE Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[14] Dahlia Malkhi, Michael K. Reiter, Daniela Tulone, and Elisha Ziskind. Persistent objects in the Fleet system. In *Proceedings of the 2nd DARPA Information Survivability Conference and Exposition (DISCEX II)*, 2001.

[15] Petros Maniatis and Mary Baker. Secure history preservation through timeline entanglement. In *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002.

[16] David Mazières and Dennis Shasha. Building secure file systems out of Byzantine storage. In *Proceedings of the 21st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 108–117, July 2002. The full version is available as NYU computer science department technical report TR2002-826, May 2002.

[17] Michael Reiter and Li Gong. Securing causal relationships in distributed systems. *The Computer Journal*, 38(8):633–642, 1995.

[18] Michael K. Reiter. The Rampart toolkit for building highintegrity services. *Lecture Notes in Computer Science 938*, pages 99–110, 1994.

[19] Sean Rhea, Patrick Eaton, and Dennis Geels. Pond: The OceanStore prototype. In *2nd USENIX conference on File and Storage Technologies (FAST '03)*, San Francisco, CA, April 2003.

[20] Rodrigo Rodrigues, Miguel Castro, and Barbara Liskov. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 15–28, Chateau Lake Louise, Banff, Canada, October 2001. ACM.

[21] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[22] Dale Skeen. Nonblocking commit protocols. In *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, Ann Arbor, MI, April 1981.

[23] Sean W. Smith and J. D. Tygar. Security and privacy for partial order time. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, pages 70–79, Las Vegas, NV, October 1994.

[24] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Mike Dahlin. Separating agreement from execution for Byzantine fault tolerant services. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 253–267, Bolton Landing, NY, October 2003. ACM.

[25] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, November 2002.

Ensuring Content Integrity for Untrusted Peer-to-Peer Content Distribution Networks

Nikolaos Michalakis   Robert Soulé   Robert Grimm
New York University

Abstract

Many existing peer-to-peer content distribution networks (CDNs) such as Na Kika, CoralCDN, and CoDeeN are deployed on PlanetLab, a relatively trusted environment. But scaling them beyond this trusted boundary requires protecting against content corruption by untrusted replicas. This paper presents *Repeat and Compare*, a system for ensuring content integrity in untrusted peer-to-peer CDNs even when replicas dynamically generate content. *Repeat and Compare* detects misbehaving replicas through attestation records and sampled repeated execution. Attestation records, which are included in responses, cryptographically bind replicas to their code, inputs, and dynamically generated output. Clients then forward a fraction of these records to randomly selected replicas acting as verifiers. Verifiers, in turn, reliably identify misbehaving replicas by locally repeating response generation and comparing their results with the attestation records. We have implemented our system on top of Na Kika. We quantify its detection guarantees through probabilistic analysis and show through simulations that a small sample of forwarded records is sufficient to effectively and promptly cleanse a CDN, even if large fractions of replicas or verifiers are misbehaving.

1 Introduction

Inadvertent and malicious content corruption by misbehaving peers is one of the largest problems in today's peer-to-peer networks. Previous work has addressed the problem in the contexts of file sharing [45, 25] and cooperative storage [10, 12, 28]; this paper addresses the problem for content distribution networks (CDNs). Existing peer-to-peer CDNs, such as Na Kika [20], CoralCDN [16], CoDeeN [46], and CobWeb [43], are currently deployed on PlanetLab, which provides a relatively trusted environment with resources donated largely by the research community. But scaling such systems to the internet at large requires protecting against content corruption by untrusted nodes. Potential attacks range from serving stale content (to save bandwidth), bypassing content transformations (to reduce CPU and memory requirements), to out-right modification, for example, by inserting ads (to generate profit).

The gravity of the problem depends in large part on whether a CDN serves only static content or both static and dynamic content. For CoralCDN, CoDeeN, and CobWeb, which serve only static content, signing the content and response headers at the origin server and verifying these signatures at the clients is sufficient to detect tampering by CDN nodes [4]. Additionally, timestamps can be used to ensure freshness. However, for Na Kika, which also serves dynamic content, hashes, timestamps, and signatures alone cannot establish content integrity—after all, the CDN nodes *are* content producers. Since web-based applications increasingly rely on the dynamic creation and transformation of content [5], CDNs need to support dynamic content and providing a general solution for content integrity becomes a crucial challenge.

This paper presents *Repeat and Compare*, a system for ensuring the integrity of both static and dynamic content in untrusted peer-to-peer CDNs. The key idea behind *Repeat and Compare* is to leverage the peer-to-peer substrate to *repeat* content generation on other nodes and then *compare* the results to detect misbehavior. Principally, this approach is comparable to the use of data replication and comparison through voting or reputation in other peer-to-peer systems, with the crucial difference that *Repeat and Compare* focuses on the replication of computations instead of data. The goal is to enable verifiable and accountable distributed computations similar to the models described in [23, 34, 50, 51].

Repeat and Compare must address three main challenges. First, it needs to observe responses sent to clients, even though clients may lie. Second, it needs to repeat response generation, even though origin servers may supply multiple, conflicting copies of the content. Third, it needs to isolate misbehaving nodes, even though the nodes that "repeat and compare" are not trusted. The underlying source for all three challenges is the same: a "he said, she said" conflict between three mutually untrusting parties: clients, replicas, and origin servers.

To eliminate the "he said, she said" problem, *Repeat and Compare* uses attestation records. Attestation records are included in responses and cryptographically bind origin servers to their content and replicas to their code, inputs, and dynamically generated output. Clients then select a random sample of records and forward them to replicas acting as verifiers. Verifiers, in turn, detect misbehaving replicas by locally repeating response generation and comparing their results with the attestation records. To isolate misbehaving replicas, the system relies on a small core of verifiers, which is trusted by both clients and origin servers, and which distributes a list of suspected replicas. If using a trusted core of verifiers is undesirable, then suspected replicas are distributed using a decentralized trust model. Under this model, each verifier maintains its own list of suspected replicas. Clients learn about misbehaving replicas by contacting verifiers with which they have off-line trust relationships.

We have implemented our system on top of Na Kika and quantify its detection guarantees through probabilistic analysis. We also show through simulations that *Repeat and Compare* effectively and promptly isolates misbehaving replicas, even if a large fraction of replicas and verifiers are misbehaving, and if only a small sample of attestation records reaches well-behaved verifiers.

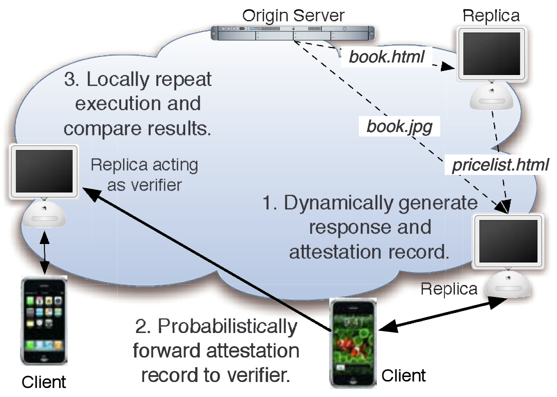
2 System Design

2.1 Requirements

There are three requirements our system must satisfy. First, users should be able to donate resources to the infrastructure without having to submit their nodes to the administrative control of a central authority. This enables a more flexible replica deployment that can cover a wider range of clients, whether they are geographically dispersed or mobile. Second, the type of dynamic content that replicas can generate should be restricted as little as possible. This enables applications to scale independent of their functionality. Previous solutions for dynamic content generation restrict proxy functionality to a small set of easily verifiable functions that modify original content through add/remove/replace operations [31]. But most of the interesting functionality in web services today involves executing general purpose scripts for content management [3]. Third, a mechanism for ensuring the integrity of content generated within the CDN must not rely on the ends, i.e., clients or servers, because it leads to a tension between scalability and trust. Relying on origin servers to verify the integrity of content delivered through the CDN does not scale. Also, it is easy and cheap to create client or server identities and use them to subvert detection.

2.2 System Overview

To illustrate how *Repeat and Compare* works, we walk through an example. A user browses through her mobile device an online bookstore that aggregates book listings put together by a large number of independent bookstores. To help scale the bookstore application, each bookstore has donated nodes to serve as replicas. Each request from the mobile device may involve transforming media to fit a small screen or sorting a list based on pricing information. If intermediate results have already been generated by other replicas (e.g., media transcoding), they can serve as inputs for generating the final response, thus saving computational overhead and bandwidth.



Origin Server
Replica
book.html
3. Locally repeat execution and compare results.
book.jpg
pricelist.html
Replica acting as verifier
1. Dynamically generate response and attestation record.
Replica
2. Probabilistically forward attestation record to verifier.
Client
Client

Figure 1: *Repeat and Compare* overview.

Figure 1 shows a high-level view of how *Repeat and Compare* can verify the operation of an untrusted CDN. Clients access content through peers in the CDN that act as *replicas* of an *origin server*. Replicas are used both as caches for static content, such as book items, as well as generators of dynamic content, such as sorted price lists of books [16, 20, 43, 46], while origin servers keep authoritative copies of the content. Replicas locally generate responses if they have all the necessary inputs. Otherwise, they contact other replicas or origin servers for inputs. In addition, replicas can cache dynamically generated content using site-specific scripts [20]. This leads to the flow of content and computations illustrated in the upper right of Figure 1.

At this point, the client has received a response with possibly corrupted or stale content. In the bookstore example, it is possible that a replica sorts search results so that certain books appear at the top to boost the sales for a particular seller. To ensure content integrity *Repeat and Compare* takes an optimistic approach: the client accepts the response, but informs the CDN about it. First, each replica embeds an attestation record in the response. Second, the client forwards the record to a randomly selected replica that acts as a *verifier*. To reduce the overhead imposed on the system, the client forwards only a sample of the records that it sees. Third, to detect misbehaving replicas, the verifier repeats execution and checks if the result matches the record sent by the client.

Verifiers, however, may be malicious. To expose misbehaving verifiers, *Repeat and Compare* uses either a centralized or a decentralized approach. By deploying a small, centralized set of trusted verifiers, *Repeat and Compare* relies on untrusted verifiers to absorb most of the verification load and on the trusted ones to produce the final result. If a trusted set of verifiers is not desirable, then, in order to learn which replicas are misbehaving, clients can select which verifiers they trust using off-line relationships.

Our design meets the three requirements as follows. Using attestation records to enforce accountability allows users to donate resources to the infrastructure without having to submit their nodes to the administrative control of a central authority. Repeating execution and comparing results enables applications to scale independent of their functionality. Organizing replicas using either a trusted core or decentralized trust relationships helps counter the tension between scaling detection and trusting the outcome of verification.

2.3 Design Space

Comparable to PeerReview [23], our system ensures content integrity through detection rather than prevention (e.g., BFT [7]) for three reasons. First, in contrast to prevention, detection does not impose a bound on the number of misbehaving replicas. As long as there is one well-behaved replica, it can detect every misbehaving replica, since it locally repeats execution. Second, detection is cheaper and can be decoupled from the actual request-response process. This is important since the main reasons for using CDNs are to reduce client latency and bandwidth consumption across the internet. Third, detection is good enough both for bounding the damage over the client population and over time. In addition, it reduces the incentives for misbehavior by exposing misbehaving nodes.

Basically, *Repeat and Compare* treats misbehavior as a failure and thus represents a failure detection system. As a result, it provides *completeness* and *accuracy* guarantees as described in [8]. Completeness ensures that misbehaving nodes will be suspected by the system. Accuracy ensures that well-behaved nodes will not be suspected by the system. To motivate our solution, we present a series of straw man approaches. We show how to overcome the shortcomings of each approach, thus justifying our design. Table 1 summarizes the main idea, benefits and drawbacks for each approach.

Detection by Clients: A natural application of the end-to-end argument [37] would recommend that the client be the verifying end, since it is also the receiving end. This approach is suitable for static content, when the content and response headers are signed by the origin server. But for dynamic content, clients cannot always verify the integrity of received data as it may be expensive to do so. In the bookstore example, a client would need to dynamically generate a price list for different books. This would involve contacting different vendor servers, which nullifies the benefits of the CDN. Another option is for clients to send multiple requests to randomly chosen replicas, similar to sampled voting [28]. However, the only benefit of using this approach is to tolerate a small fraction of misbehaving replicas. If a large fraction of replicas is misbehaving (i.e. more than half), then voting provides neither accuracy nor completeness.

Detection by Spies: An alternative for detecting misbehavior is to have replicas imitate clients and send requests to other replicas to monitor their behavior. Since replicas can misbehave at will, if replica identities are public, direct replica-to-replica monitoring will not work. One way to overcome this problem is to use hidden verifiers. The idea is that a trusted party deploys a set of "spies", nodes whose identities are secret and which imitate regular clients. Detection through spies is attractive because it does not require any modification to the application protocol, including the clients and servers using the CDN.

However, spies are not a feasible long-term solution. Since spies do not know when misbehaving replicas send corrupt responses, they must maintain a constant flow of monitoring requests to detect misbehavior. As a result, it is very likely that, after some arbitrarily long period, every spy that monitors a particular replica will have sent at least one request to that replica. A malicious replica can patiently serve correct responses until a certain timeout, while recording client addresses in a "suspected spies" list. Then it can serve corrupt responses to clients not on the list. Using this strategy, it can remain undetected with high probability. Spies do not have a way to detect that they are suspected because not observing corrupted responses does not imply that a spy was caught. After all, it is also possible that there are no misbehaving replicas in the system. So, either new spies need to be periodically deployed or existing spies need to replenish their network addresses, even if there are no misbehaving replicas. Although this solution ensures accuracy, it is not attractive because in order to provide completeness it leads to an arms race for maintaining spies.

Detection by Informers and Reputation: The alternative is to use responses received by clients. The basic idea is that some clients act as "informers" and volunteer to forward received responses back to a verifier. The verifier detects if a replica is misbehaving by locally generating the response and comparing it with the forwarded response. This idea is attractive because it requires only volunteers to make it work on the existing web. No changes to servers are necessary and, except for the volunteers, clients remain unmodified. However, using informers without protecting the content of responses leads to the "he said, she said" problem. If a verifier sees a corrupt response, it does not know if it came from a replica or was tampered with by the client.

Using a reputation system to compute an aggregate trust score for informers can lessen the "he said, she said" problem by limiting the effects of dishonest informers [25]. A verifier decides what the untampered

Approach	Key Idea	Benefits and Drawbacks
Detection by Clients	Clients verify the integrity and freshness of the data they receive.	Protects static content. Resource-limited clients may not be able to verify integrity of dynamically generated content.
Detection by Spies	A set of spies imitates clients by sending requests and then verifying responses.	No changes to clients or servers, but cannot know if a spy is compromised.
Detection by Informers & Reputation	Verifiers use responses forwarded by clients to compute a replica's trust score.	Relies only on volunteers, but suffers from the "he said, she said" problem.
Detection by Trusted Verifiers using Attestation	Replicas held accountable through attestations. Response verification is performed by a small set of trusted verifiers.	Solves the "he said, she said problem". Does not scale.
Servers as Verifiers	Origin servers verify a sample of the responses using records from informers.	No need for globally trusted verifiers. Overloads popular servers. No ownership for aggregated content.
Detection by Untrusted and Trusted Verifiers	Relies on untrusted verifiers to assist trusted ones in verification.	Scales verification in the common case of correct responses by reducing the probability of detection.
Decentralized Trust	Verifiers keep local lists of misbehaving replicas. Clients learn about replicas using offline trust relationships.	No need for globally trusted verifiers. Detection is slower.

Table 1: Overview of potential approaches for detecting misbehaving replicas, listing the key idea, benefits and drawbacks for each approach.

response is, based on a weighted majority vote from different informers, before it repeats the response generation. In a large-scale environment, however, a reputation system can be gamed by a strong adversary that can gain a large enough foothold inside the volunteer informer group. Large botnets spanning hundreds of thousands to millions of machines make this threat real [47]. As a result, this approach does not provide completeness nor accuracy.

Detection by Trusted Verifiers using Attestation: If responses are signed by replicas, then malicious informers cannot tamper with them, so an informer-based reputation system is not necessary for resolving the "he said, she said" problem. By attesting to the integrity of their responses, replicas are held accountable. In addition, attestations must be linked back to the original data and code, so origin servers must also attest to the integrity of their original data. In the bookstore example, if a book cover image is downsized to fit a handheld, it must be linked back to the original image. So, forwarding attestation records to verifiers is sufficient for ensuring completeness and accuracy.

The next issue is deciding who verifies attestation records. The simplest approach is to use a small set of verifiers that are trusted by both clients and origin servers. However, when most of the records are valid, i.e., they correspond to correct responses, trusted verifiers can be overwhelmed, since the forwarded records grow in proportion to the client population. Because replicas generate content that might depend on client properties or the time of the request, using a cache for consistent records will not prove useful. Even if clients forward only a fraction of received records, verifiers must still handle traffic that is proportional to the number of clients. As a result, trusted verifiers are a bottleneck in the common case and a better verification approach is necessary.

Servers as Verifiers: An alternative is to use origin servers as trusted verifiers for their content. The idea is that clients forward records to origin servers only for responses related to their own content. This is attractive because clients and servers are not required to trust a third party to perform verification. Servers could deny access to their content to replicas that they consider misbehaving. However, this approach has three problems. First, servers need to verify responses proportional to the popularity of the content and, as a result, this approach does not scale well. Second, clients must somehow learn which replicas are misbehaving or not, so that they avoid them. This means maintaining per-server state at the client. Third, in collaborative applications, no single server has ownership of the delivered content. In the example of the collaborative bookstore from Section 2.2, a list of book prices is generated from content that belongs to different vendors. As a result we dismiss this approach.

Detection by Untrusted and Trusted Verifiers: A way to scale verification that does not rely on clients or servers is to filter valid records by redirecting verification to untrusted verifiers. Untrusted verifiers then forward only invalid records to trusted verifiers (if they forward valid ones they are misbehaving). This solution scales for two reasons. First, since all replicas can perform the same functionality, we can have as many verifiers as replicas in the CDN. Second, a single invalid record is a proof of misbehavior, so the total number of records processed by trusted verifiers is on the order of the number of misbehaving replicas at any given time. Because attestation records are tamper-evident, the worst an adversarial untrusted verifier can do is drop the record of a corrupt response, thus delaying detection. However, if every record seen by a client is forwarded to a verifier, the total traffic to the CDN is essentially doubled. Since using untrusted verifiers already provides only probabilistic completeness, this motivates an approach based on sampled forwarding. Clients only forward an attestation record with probability p.

Decentralized Trust: A globally trusted set of verifiers greatly reduces the communication and computational overhead of detection because results can be publicized and believed by everyone in the system. However, it is not always desirable to have a globally trusted authority, especially when a CDN is built through collaborative efforts and is self-managed [20]. Instead, a decentralized detection mechanism is necessary, where each misbehaving replica is independently detected by each correct replica. Under this model, clients learn about misbehaving replicas by contacting verifiers with whom they have off-line trust relations. To ensure completeness, each misbehaving replica must be detected by all correct verifiers/replicas. As a result, the computational overhead per misbehaving replica is on the order of the number of replicas in the system—after all, each replica will have to verify attestation records by itself.

Decentralized detection may be augmented using gossip initiated either by clients or verifiers. A client can randomly select v verifiers to forward an attestation record, thus increasing the chances that it reaches a “good” one by a v:1 ratio. However, it also imposes a v:1 increase in traffic in the CDN, which is not desirable. Gossip might be more beneficial when used by verifiers. Since verifiers only forward incriminating attestation records to other verifiers, the communication and computational overhead in this case is the same as if each verifier was contacted by a client instead.

Summary: We have explored the design space in order to successively pin down *by whom* and *how* detection is performed. As a result, we deduce three core properties for *Repeat and Compare*. First, solutions that preserve existing servers and clients are not sufficient. Servers must sign their responses and clients must verify them. In addition, replicas must produce attestation records that clients can forward to replicas, which act as verifiers and thus detect misbehavior. Without the accountability enforced by attestation records, there are no detection accuracy guarantees. Second, if clients forward all records they observe, then traffic is essentially doubled. This motivates a sampled forwarding approach, which provides only eventual detection completeness with high probability. Third, there is a trade-off between trust and computational cost. When there are trusted verifiers, the cost for detection is low. When trusted verifiers are not desirable, decentralized detection can offer the same guarantees, but with a cost proportional to the size of the CDN.

2.4 Repeat

Repeat essentially simulates the response generation process illustrated in Figure 1 by recursively generating intermediate results until it outputs the final response. Repeating execution in the presence of non-determinism, external inputs and implicit parameters might not always produce identical results. Removing non-determinism for general purpose computing is impossible. However, web-based architectures are restricted enough to make the task tractable. Client requests are processed independently through well-defined interactions at the HTTP protocol level and applications are relatively stylized, written in scripting languages with bounded functionality.

Identifying external inputs (client request, original content) and configuration parameters (server configuration, library versions) is as important as identifying which code to repeat. Repeating the same code with different inputs, versions of inputs, or configuration parameters will likely produce different outputs. Therefore, external inputs and configuration parameters are uniquely identified by a name, timestamp, and value. They are then cryptographically bound to the code that used them via attestation records. Replicas use this explicit information to set up a runtime equivalent to the one that generated the original response.

Attestation Records

As mentioned earlier, attestation records are a necessary ingredient to preventing the “he said, she said” problem caused by untrusted origin servers and clients. What allows an observer to detect misbehavior using attestation records is a chain of accountability from the client to the server. A broken chain serves as a misbehavior detector, whereas a solid chain proves compliance with the protocol.

Attestation records are bound to either *literals* or *references*. Literals are explicit data values such as system clock times, seeds for random number generators, sensor values etc. They are typically small inputs that come either from origin servers or clients. References are uniquely named data that need external communication to be fetched. These are data that have been dynamically generated and labeled by replicas as part of the response generation process. Examples include transcoded media and aggregated content. In addition, static content that resides on origin servers is treated as a reference. Although such content could be represented by literals, using references saves space in the attestation record in case the content is large.

In order for accountability not to be compromised, a record must have the following properties:

- It must bind a replica or origin server to the response it generated, so that it cannot later deny it generated that response.
- It must be practically infeasible to generate a record that incriminates another replica.
- A verifier must be able to verify the integrity of the response using only the record and its own execution environment.

To show how attestation records satisfy those properties we explain how they are structured. Literals are of the form

(TYPE, TIMESTAMP, VALUE, PUBKEY, SIGNATURE)

and references are of the form

(TYPE, NAME, TIMESTAMP, VALUE, INPUT-LIST, PUBKEY, SIGNATURE).

In more detail, the different attestation record entries have the following roles.

TYPE. For literals we distinguish types into CONFIG, REQUEST, and ONETIME because each is processed differently. CONFIG records are used to set configuration parameters and initialize the runtime environment. One example is setting pseudo-random number generator parameters such as the algorithm or seed. REQUEST records bind the requester to the generated content and are sent together with the request; they ensure that a client cannot lie about the content it requested. ONETIME records are discussed below. References always have type RESPONSE, since they represent the content returned with a response. The difference between static and dynamic content is determined by checking for an empty input list.

NAME. The name is a unique reference to the content, a URL in our system. Since URLs embed the name of the authoritative server, a replica cannot claim authorship of content it does not own. Names are not necessary for literals, which contain the actual data as part of the value entry.

TIMESTAMP. The timestamp identifies different content versions and is used to ensure the freshness of a response in case content has been updated by the origin server. Its use is explained in more detail in the next section.

VALUE. The value is used to verify integrity; it must match the content produced during verification. For references, the value is a digest of the response. For literals, it is the actual plain-text value. For REQUEST records, it contains client-specific inputs that affect how content is processed (e.g., `User-Agent` for transforming images for cell phones).

INPUT-LIST. An optional list of attestation records declaring the inputs used to generate a record of type RESPONSE. The list contains a record for the executable, configuration parameters, and any external inputs. Using input lists, the RESPONSE record is sufficient to guide the verifier through execution. Notably, the structure of the record resembles the recursive process of content generation in a CDN and thus provides the verifier with all the information necessary for generating the final response from scratch. Note that, besides a constant number of input list entries to identify a replica's execution environment and content processing script, the length of an input list is proportional to the number of aggregated HTTP resources.

PUBKEY. The public key is bound to the identity of the principal that this record speaks for. Any of the existing distributed infrastructures [14, 54] can be used for managing public keys. The certification authority plays no role in the CDN itself.

SIGNATURE. A signature using the principal's private key. It signs all other entries of the attestation record. The signature binds an origin server to its content. It also binds a replica to the original request, the replica's execution environment, any inputs, and the final response. Assuming that an adversary cannot break cryptographic primitives, one must steal a replica's private key in order to incriminate it.

Freshness

To ensure freshness, the client must receive the *latest* version of the content as defined by the origin server. Since both servers and replicas can lie, to determine the latest version, servers must promise not to modify content for a predetermined time period by setting the TIMESTAMP of the RESPONSE record to an expiration time in the future. This ensures that there is only one latest version, the one with a timestamp that has not expired. When a request is made, the client sets the current time as the TIMESTAMP of the REQUEST record. To determine the current time, we assume that all nodes (clients, replicas, origin servers) have loosely synchronized clocks, i.e., all clocks are within skew σ and rates do not diverge. To ensure that clocks remain loosely synchronized, our system trusts an external time service such as NTP. To detect that a server sent multiple conflicting versions, it is sufficient for a verifier to see two records signed by the server with versions greater than the timestamp of the request (within σ).

Since there is no third party that can observe when a client sent a request to a replica or when a replica sent a request to an origin server, ensuring freshness requires addressing a time-dependent instance of the "he said, she said" problem. If a client forwards a record at time t that contains a timestamp e, where $e < t$, the verifier cannot distinguish whether the replica sent stale content or the client waited until time $t > e$ before forwarding the record. To address this issue, each requester and responder agree on the time of the request through the use of the REQUEST record. The responder accepts the request only if the embedded time is equal to its own time within σ. Since the request record is part of the response record's input list, it is signed by the replica, so the client cannot modify it. Upon receiving the response, the client checks that an attestation record's request time was not modified by a malicious replica.

With this information, a verifier can detect if content is fresh by using the attestation record to check that (a) the timestamp of the request is less than the timestamp of the response plus σ, and that (b) the timestamp of the record is less than the timestamp of each input plus σ. In addition, the verifier needs to perform a sanity check to avoid errors caused by clock skew. It needs to verify that the difference between its current time and the timestamp of each input is greater than σ before it verifies freshness. This check ensures that every node has either transitioned to the newer version of the content or still sees the current one.

Non-Repeatable Data

When repeating dynamic content generation, care must be taken to identify which data intrinsically change over time and, as a result, are not repeatable. We identify two types of such data: one-time values and randomly generated data. One-time values depend on the exact time that they were produced. Examples include sensor measurements and stock prices. The problem with repeated execution is that a verifier repeats the response generation process at a later time, so if it makes a request to a content producer for a one-time input it will get a different value. We avoid this issue through the use of ONETIME literals. We restrict their generation to origin servers because the integrity of these records cannot be verified otherwise.

Similarly, when data are randomly generated, repeating the process will result in different outputs with high probability. To avoid this issue, applications are constrained to use pseudo-random number generators and produce a CONFIG literal with the seed inserted in the VALUE entry. To defend against attacks where a replica "fixes" seeds to reduce randomness, seeds can be provided either by origin servers or clients.

Runtime

Before a verifier repeats execution, it is important to set the correct runtime environment. This involves loading the correct code and server configuration. We use CONFIG records for that purpose. These records are bound to the version of the executable, dynamically loaded libraries, and server configuration files. For the executable and the libraries, the records are bound to a digest of the binary. For configuration files, they are bound to specific parameters that must be the same across all replicas. Scripted code executed by a replica, such as Javascript in the case of Na Kika [20], is supplied as part of the RESPONSE record's input list. If any of the libraries or configuration parameters do not match the records, then the verifier treats a replica as misbehaving.

2.5 Compare

Compare consists of two stages: forwarding attestation records to verifiers and then detecting misbehaving replicas. As explained in Section 2.3, attestation records are forwarded to verifiers via clients. A client first checks that each response contains a consistent attestation record; otherwise, the client drops the response. Then it forwards the record with probability $p < 1$ to a randomly selected verifier. Sending a fraction rather than all the records helps reduce the traffic overhead in the CDN. Because current web clients do not support the notion of attestation records clients either have to be modified or install a local web proxy [33]. Unless a large enough fraction of the client population supports attestation records, replicas have no incentive to produce attestation records. To detect replicas, we adopt both centralized and decentralized approaches.

Centralized Detection

As explained in Section 2.3, if there is an authority that is trusted by both clients and origin servers, then this authority can deploy a small set of trusted verifiers to detect misbehaving replicas. To scale detection, we leverage the massive replication offered by the CDN. Untrusted verifiers receive records from clients, perform *Repeat* and if they detect a misbehaving replica, they forward the incriminating record to a trusted verifier. To prevent malicious verifiers from overloading trusted ones, if the forwarded record is not incriminating, then the untrusted verifier is considered misbehaving and punished (same as any misbehaving replica). As a result, misbehaving verifiers have no incentives to forward "good" records and the traffic seen by trusted verifiers is proportional to the number of misbehaving replicas. By definition, trusted verifiers are trusted by all nodes in the system. Consequently, once a replica has been detected as misbehaving, no further verification of that replica's responses by other replicas is necessary.

Decentralized Detection

If a globally trusted authority is not desirable, then detection can be performed using a decentralized transitive trust model similar to Credence [45]. In Credence, each peer keeps a local voting table for files it has received. It uses the table as a guide to find like-minded peers through statistical correlation of votes. Our model differs from Credence in two ways. First, votes are not cast by manually examining content as in file sharing, but through *Repeat*. Second, the votes are not about content, but about replica status, i.e., misbehaving versus well-behaved.

So, decentralized detection works as follows. Like in the centralized model, replicas act as verifiers. Clients forward attestation records to them and they locally repeat execution to detect misbehavior. If they detect misbehavior, however, they do not forward the attestation record to trusted verifiers as before. Instead, they maintain a local voting table that is initialized with positive votes for every replica. If they receive an attestation record that incriminates a replica then they permanently change their vote for that node to negative. Clients access the CDN by contacting a *friend*, a verifier that they trust through off-line trust relationships. Friends then forward to the clients lists of replicas that they believe are trustworthy, i.e., rank on the top of their lists. Clients use the lists to find nearby replicas to access content. They refresh the list of replicas by periodically contacting their friends.

Punishment and Redemption

Once replicas are detected they are punished. A strict punishment policy would require that replicas be banished. However, this might be too harsh in practice as misbehavior may be the result of human error. We acknowledge that there is no perfect solution to this problem. But, since our system decouples detection from punishment, a real deployment can use application-specific policies to let nodes reenter. For example, in the bookstore application of Section 2.2, one can use tit-for-tat punishment: If a replica donated by a particular vendor misbehaves, replicas block access to the vendor's content (i.e., remove the vendor's books from the price list) for a few hours to decrease sales.

Detection Guarantees

As explained in Section 2.3, failure detectors (as defined in [8]) are characterized with respect to *accuracy* and *completeness*. In terms of accuracy, *Repeat* ensures that no well-behaved replicas are ever suspected, no matter how many records are sent to verifiers. Completeness, however, depends on *Compare* and is only probabilistic. For every corrupt response sent by a misbehaving replica, there is always a positive probability that the replica will be detected.

Eventual completeness is guaranteed only if the system maintains randomness. If clients keep forwarding attestation records to the same verifiers, some misbehaving replicas may never be detected. This is a real possibility for CDNs because clients usually get redirected to nearby replicas. Other than locality, however, completeness is independent of the underlying peer-to-peer topology; as a result, *Repeat and Compare* works with arbitrary peer-to-peer CDN architectures. In practice, it is important to quantify how fast the system detects misbehaving replicas and how much damage they can realize before detection. We quantify *Repeat and Compare*'s eventual completeness guarantees in Section 4.1.

3 Implementation

Our prototype implementation of *Repeat and Compare* builds on four open source packages: the Apache 2.2 web server, the Apache-based Na Kika CDN [20], the Privoxy personal proxy [33], and the OpenSSL cryptographic library [30]. Server- and replica-side functionality is provided by two Apache filters. One filter signs content and comprises 1,000 lines of C code, while the other filter verifies content and comprises 700 lines. Client-side functionality is provided by a 600 line extension to Privoxy, which decouples our implementation from the actual browser used to access the CDN. All three components leverage an attestation record library, which builds on OpenSSL and comprises 4,400 lines of C++ code.

For Na Kika, we modified the *fetchURL* function to add remotely accessed inputs to an attestation record's input list; we also added hooks to get and set the seed of JavaScript's random number generator. Our attestation record library exports a C/C++ API for creating, parsing and validating attestation records using RSA signatures. Content digests are computed using SHA-1. Attestation records are passed between nodes as `X-Attestation` response headers. The configuration and the runtime information is expected to change slowly. So, to amortize the cost per response, it is periodically updated using hashes of the Apache executable, libraries, and configuration files. Replicas simply insert the hashes as inputs in each attestation record. Finally, for the signer module we implemented an in-memory cache to store records for content that have already been hashed and signed.

4 Evaluation

The hypothesis of this work is that *Repeat and Compare* makes fast forward progress at detecting misbehaving replicas even if a very large fraction of replicas are misbehaving. Detection depends on the forwarding probability p, the number of misbehaving and well-behaved replicas in the system, and the number of corrupt responses a misbehaving replica has sent. To quantify expected system behavior, we analyze how each parameter affects the probability and rate of detection, the time it takes to cleanse the system, and the damage incurred until cleansing is complete.

In practice, network and processing delays also affect the time to detection. To account for these parameters, we simulate a large-scale deployment of both the centralized and decentralized versions of *Repeat and Compare*. Additionally, locality can have a negative impact on our system's effectiveness, as it reduces randomness. In particular, CDNs usually redirect clients to nearby replicas in order to minimize client-perceived latency—but, as a result, they also limit the set of replicas and verifiers visible to each client. We account for the effects of locality through additional simulations.

Since *Repeat and Compare* requires signing all messages and repeating execution, detection can be expensive and affect a CDN's scalability. As a final assessment, we perform a set of micro-benchmarks to experimentally determine how throughput changes when servers generate attestation records and when replicas verify them.

4.1 Detection

Analysis

The goal of our analysis is to abstract away application-specific properties of CDNs and show the core guarantees of *Repeat and Compare* given perfect randomness, and no network and processing delays. We examine a static view of the system, where no new nodes enter or leave (other than those detected). We assume that clients forward records with probability p and that there are f misbehaving and g well-behaved replicas in the system. g is constant over time because our system ensures accuracy. Since a client will receive corrupt content with probability $f/(f+g)$, content integrity is ensured when all misbehaving replicas have been detected. So, we are interested in four properties of our detection mechanism: (a) the probability of detecting a misbehaving replica, (b) the rate of progress made by detection, (c) the time it takes to detect all misbehaving replicas and (d) the damage done (number of corrupted responses) until all the misbehaving replicas are detected.

The probability P_D that a misbehaving replica is detected depends on (1) the client forwarding the attestation record and (2) the client forwarding it to a good verifier. Given a sample of attestation records sent by a replica, P_D depends only on b, the number of incriminating (bad) records sent (good records have no effect). To get a lower bound on P_D, we fix f during the time that b records are sent. For every record sent, the probability that it reaches a good verifier is:

$$P_V = \frac{pg}{f+g}, \text{ therefore } P_D = 1 - (1 - \frac{pg}{f+g})^b$$

If f is small, then P_D improves as p increases, otherwise a large f masks the effects of p. However, even if $p = 1$, P_D can reach 1 only by increasing b.

To analyze the forward progress in detecting misbehaving replicas, we examine how the rate of detection changes dynamically as the misbehaving replicas are detected and removed. To simplify the analysis, we assume that at each time unit each replica serves one client response and that clients contact verifiers randomly. Since inactivity does not affect detection nor corruption and replicas typically have similar capacity, we believe this is a reasonable assumption.

The rate of detection r depends on the number of bad requests sent per unit time, which is equal to f:

$$r = \frac{df}{dt} = -f\frac{pg}{f+g} \qquad (1)$$

Since detected replicas are removed, the rate is negative with respect to f. To give an insight of how this affects f over time, we examine how r changes as f decreases. We identify two interesting cases: (a) when f is a large multiple of g, and (b) when f is a fraction of g. As long as f is a large multiple of g, then $r \simeq pg$. Since both p and g are constant, detection makes steady progress. As f decreases and approaches g, the detection rate decreases proportionally to how f decreases. But, even in the worst case, where $f = 1$, the rate approaches p.

To estimate the time T it takes to detect all existing misbehaving replicas, we compute the time it takes to detect all of them except the last by solving the differential equation 1 and then compute the average time to detect the last one using P_V. The reason for not using Equation 1 to compute the total time is that when $f < 1$, t asymptotically reaches infinity as f goes to 0, so it is of no practical use. Our estimate is a lower bound because we do not take into account network or computational latency. Misbehaving replicas are detected and removed within the same time unit they served a corrupt response. We also assume there are F misbehaving replicas initially and during detection no new replicas enter the system. Solving Equation 1 gives:

$$t(f) = \frac{F-f}{pg} + \frac{1}{p}\ln(\frac{F}{f}) \qquad (2)$$

So, the total time to detect all misbehaving replicas except the last is:

$$\frac{F - 1 + g\ln F}{pg}$$

To detect the last one it takes on average an additional time:

$$\frac{1}{P_V} = \frac{g+1}{pg}$$

This is because the probability of detection follows a geometric distribution with probability P_V. So, the total time of detection is:

$$T = \frac{F + g + g \ln F}{pg}$$

For large F, $T \simeq F/(pg)$, so the time it takes to remove F misbehaving replicas depends on how large is F compared to the constant pg. For small F, $T \sim 1/p$ so detection is faster for higher values of p.

Given f misbehaving replicas in the system, the damage done by a replica until it is detected (given that it is the first one detected among f) follows a geometric distribution and is:

$$B_1 = \frac{1}{p(\frac{g}{f+g})}$$

To compute, however, the total damage B in terms of corrupt responses sent during time T, we must integrate Equation 2 from 1 to F, because f depends on t. Then, we add the total number of corrupt responses sent by the last misbehaving replica, which is T:

$$B = \frac{F^2/2 + gF + 1/2}{pg}$$

If F is large, then $B \simeq F^2/(2pg) + F/p$. So, the damage incurred is basically affected by F. If F is small, then $B \sim 1/p$. So, the damage is reduced as p increases.

The analysis shows that, if the number of misbehaving replicas is a fraction of the well-behaved replicas, increasing p helps increase the probability and the rate of detection and bound the damage. Setting a high p, however, also increases the traffic overhead by a fraction equal to p, as explained in Section 2.3. In particular, if p approaches 1, then traffic essentially doubles. But as the analysis shows, when f is large, the detection rate is dominated by the number of malicious replicas in the system. In that case, detection has enough forward progress momentum to allow for smaller values of p.

Simulations

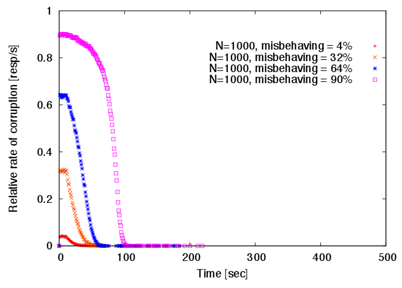
To take into account network characteristics and processing delays, we simulate *Repeat and Compare* using Narses [19] with a topology based on the Meridian all-pairs wide-area latency matrix [42]. We selected 1,000 random nodes as replicas and another 1,000 random nodes as clients. Simulation results are averaged over 10 runs. The request load per replica was set to 1.2 requests per second based on informally gathered statistics from CoralCDN's deployment [15]. Finally, the simulation abstracts away the internals of repeated execution and uses a 5 second computational delay for verifying content. It also limits each verifier to 10 concurrent requests. Both the computational delay and throughput limit are pessimistic when compared to our microbenchmarks in Section 4.3.

Figure 2 shows how long it takes to detect misbehaving replicas for various fractions of misbehaving replicas with $p = 0.1$ both in the centralized and decentralized models described in Section 2.5. Using a small set of trusted verifiers (size=4, Figure 2(a)), misbehaving replicas are detected much faster than in the decentralized detection model (Figure 2(b)). Using gossip does not seem to expedite detection (Figure 2(c)). In our simulation, each verifier forwards incriminating records to 20 randomly chosen verifiers and each forwarded record has a TTL of 2. The reason for gossip's ineffectiveness is that a verifier is more likely to get an attestation record directly through a client than through gossip due to the computational delay of verification. Overall, all three schemes take less than 500 seconds to isolate all misbehaving replicas even if they are 90% of the replica population.

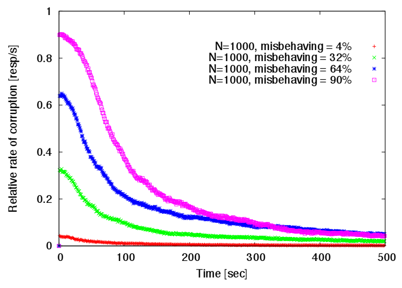
4.2 Locality

To examine the impact of locality, we use the same simulation environment as before. Since the Meridian matrix contains network distances for each pair of nodes, we use this redundant information to simulate an application-level anycast service by organizing replicas in concentric rings centered at each client starting at 2 ms and doubling the radius at every step. This is similar to how Meridian nodes organize their neighbors [48].

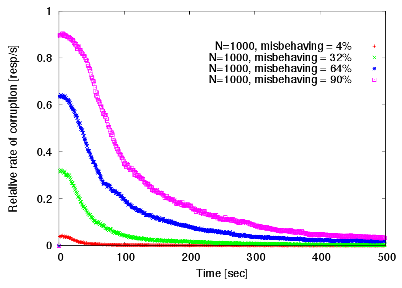
As explained before, there is a tension between locality and randomness. If the application-level anycast service is perfect and always returns the closest replica to the client, then no misbehaving replica is ever detected. This is because the same replica that serves content is also selected as a verifier. So, any incriminating attestation records will always be verified by the very same replicas that generated them. If, however, locality is relaxed by selecting a random node within a 2 ms radius from the client, then detection commences similarly to the random case, as shown in Figure 3. Relaxing perfect locality either when choosing replicas or verifiers is sufficient to ensure progress. There is a subtle difference between the two alternatives. When replicas are chosen randomly but clients remain fixed to the same verifier, the effect is the same as having each verifier scanning its proximity for misbehaving replicas (Figure 3(a)). When verifiers are chosen randomly, but clients remain fixed to the closest replica, then if a misbehaving replica is not the closest node to any client, it will never be detected. Since it is not the closest to any client, it will never serve any requests either, so it cannot do any harm (Figure 3(b)). Finally, a hybrid approach, where both replicas and verifiers are chosen randomly within the proximity of the



(a) Centralized detection.

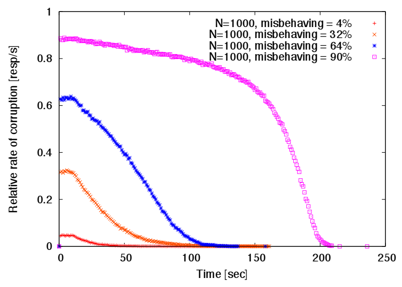


(b) Decentralized detection.

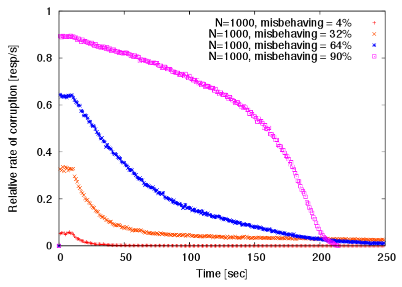


(c) Decentralized with gossip.

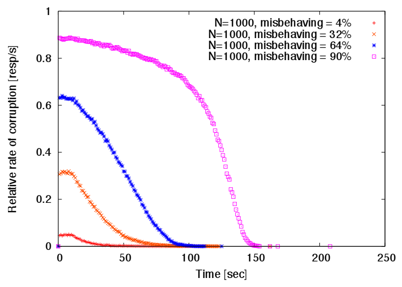
Figure 2: Comparison of centralized detection, decentralized detection, and decentralized detection with gossip.



(a) Relaxed locality for verifiers.



(b) Relaxed locality for request processing.



(c) Relaxed locality for both.

Figure 3: Locality vs. randomness. By default, clients are redirected to the closest node; for relaxed locality, a random node within a 2 ms radius from the client is selected.

client detects misbehaving replicas faster (Figure 3(c)).

4.3 Overhead

Using our prototype implementation, we evaluate the overhead of producing attestation records at origin servers and verifying them at Na Kika replicas.

Content Producer Overhead

To characterize the overhead of producing attestation records, we compare a server's throughput and 90th percentile latency when it reaches maximum capacity with and without running the signer module that produces the `X-Attestation` header. The server machine is a 3.2 GHz dual core Pentium D with 4 GB RAM running Fedora Core 6 Linux with a 2.6.18 kernel. Load is generated using the *httperf* [29] web load generator by fetching a single static 2,097 byte document representing Google's home page (without inline images). Since static resources are already well-served by existing web servers, this benchmark illustrates the worst case overhead scenario to content producers that adopt *Repeat and Compare*. Without the signer module the server reaches 1,070 responses per second (rps) with 1 ms 90th percentile latency. With the signer module, the server reaches 443 rps with 2.8 seconds 90th percentile latency. Generating attestation records adds significant overhead to the content producer because it signs each attestation record with its private RSA key. When the signer module caches records for content already signed, the server reaches 1,070 rps with 1 ms 90th percentile latency, i.e. no perceived overhead.

Verifier Overhead

To characterize the overhead of verifying attestation records, we compare the throughput of a Na Kika node running at full capacity with and without running a verifier module. The node dynamically scales an image to fit a 176 x 208 pixel cell phone screen. With the verifier module, the node also repeats execution, generating the same reduced-size image and comparing its digest with the attestation record's digest. The server machine is the same as in the previous benchmark. Na Kika and the verifier run on a 2.8 GHz Intel Pentium 4 PC with 1GB RAM running Fedora Core Linux with a 2.6.9 kernel. Load is generated as in the previous benchmark, but in addition an equal load is generated for verification. Since verification involves checking the consistency of the attestation record using the signer's public key, serving a CPU-consuming page provides a pessimistic estimate of the verification overhead. As a base, running an executable that transforms the image in a tight loop achieves 13.1 transformations per second. Without signing or verification the Na Kika node can serve 6.8 rps. We attribute the extra overhead to Na Kika's Javascript-based pipeline, which is also a large CPU consumer. When signing is enabled, the node reaches 5.9 rps. When signing and verifying simultaneously, the throughput drops to 2.8 rps. This is expected, since the verifier duplicates the response generation process, thus taking up an equal amount of resources.

5 Related Work

The typical solution for preventing message tampering between clients and servers is end-to-end connection-based encryption using, for example, the SSL protocol. This negates the functionality of the CDN, though, since replicas cannot act on the message. Merkle tree authentication [4] is a proposed alternative, where static cacheable content is signed at the server and verified by the client. Also, digital rights management schemes enable consumers to download content from untrusted distributors [1]. For dynamic content, [9, 31] propose the use of XML-based rules for adding/removing/replacing content, which is limited enough to be easily verified by a client.

Attestation has been used in the context of trusted computing to prove to a third party that a node executes trusted code [32]. The idea is to generate proofs of trust using an attestation chain that cryptographically binds an executable, the data it manipulates, and the underlying operating system to a trusted processor. However, current approaches are limited to attesting to the integrity of the executable's identity, not its functionality. As a result, they cannot detect exploits or misconfigurations (induced by human factor) [18, 39]. Ongoing work [41] focuses on providing more expressive attestations using trusted reference monitors. Since attestations incorporate the operating system, the trusted code depends on a particular OS version, making it harder to accommodate security updates as well as bug fixes and creating a tension between complete attestation and timely upgrades. Our system avoids this tension by focusing only on observable differences. Comparable to our system, the Common Language Infrastructure's *strong names* are used to uniquely identify assemblies, distinguish versions, and provide integrity [13]. Strong names consist of a text name, version number, culture information, a public key, and a digital signature similar to our attestation records.

Minimizing trust placed on nodes in peer-to-peer networks through decentralized reputation systems has been used for file sharing by computing either local values [45] or global values [25], both based on user ratings. As mentioned in Section 2.3, a reputation system can be used for CDNs in order to replace trusted verifiers. However, it is useful only when combined with attestation records and applied to replicas, not clients. Also, sampled voting for content integrity has been used in the context of the peer-to-peer data preservation system LOCKSS [28]. LOCKSS peers verify data integrity by collecting votes from a sample of the population and comparing them with their local copy. Sampled voting combined with repeated execution could provide clients with information for detecting misbehaving replicas without the need for verifiers. However, using verifiers is more suitable for CDNs because unilateral policies for punishing misbehavior are easier to build even if the majority of replicas are misbehaving.

The effects of misbehaving peers can be nullified using byzantine fault tolerance as described in PBFT [7] if the number of misbehaving peers is less than 1/3 of the total number of nodes. However, in a collaborative peer-to-peer CDN, there could be an unlimited number of rational users that donate nodes to benefit from the use of the CDN, but deviate from the service to gain a free ride. This behavior has been modeled as Byzantine-Altruistic-Rational (BAR) [2]. Using the layered approach proposed in [2] for building BAR tolerant systems, one could provide stronger integrity guarantees than our system because *Repeat and Compare* cannot produce proofs of misbehavior when replicas deny service to clients. At the same time, by detecting misbehavior after the fact instead of preventing it altogether, *Repeat and Compare* requires only one replica to generate a response and at most two well-behaved replicas to verify the response (in our centralized model). As a result, *Repeat and Compare* can maintain lower latency and higher throughput than PBFT and BAR systems.

Using repeated execution to detect misbehavior has been used in the context of bug discovery in Rx [35] and worm containment in Vigilante [11]. Our approach differs in that execution is repeated by a separate party, the verifier, rather than the client. Repeated execution in Pioneer [38] is closer to our approach because a trusted platform, the dispatcher, verifies the integrity of code running on an untrusted platform. But since generating a proof of correctness in Pioneer is extremely time-sensitive, it is not suitable for large scale systems.

Our system can benefit from an application-level anycast service that can serve as a controlled entry point for peers that wish to join the CDN. We believe it would be easy to modify systems such as OASIS [17] or Meridian [48] to also redirect clients to verifiers. *Repeat and Compare* has already been integrated with Na Kika [20], which uses a structured overlay to coordinate caches. Structured overlay networks provide robust and scalable coordination strategies [21, 22, 27, 36, 44, 52] and have been successfully used for static content distribution in CoralCDN [16] and Squirrel [24]. However, our system is independent of the coordination mechanism and can also be used by systems such as CoDeeN [46], ColTrES [6], Tuxedo [40] and DotSlash [53] that use domain-specific topologies and algorithms to balance load and absorb load spikes.

The problem of verifiable, accountable distributed computations has been explored in the contexts of secure information aggregation in sensor networks [34], byzantine fault detection [23], networked services [50, 49], and commercial peer-to-peer computing [51]. Comparable to our approach, these efforts hold both clients and servers accountable for their statements, using some form of attestation record. In departure from our approach, several efforts employ deterministic detection strategies, providing stronger guarantees but requiring more resources. One significant difference (and contribution) of our work is the systematic exploration of the design space.

6 Discussion and Future Work

Comparing the role of determinism in fault detection and fault prevention is instructive. Fault prevention requires that failures are independent and thus introduces non-determinism through n-version programming and dissimilar hardware components. In contrast, fault detection requires that all observable replica behavior be deterministic and repeatable. This limits detection systems, such as ours, to services that do not depend on true randomness. Our implementation meets this requirement by ensuring identical executable versions and runtime environments, including the use of pseudo-random number generators and identical seeds. It is an open issue whether this approach generalizes beyond web applications. We believe that virtual machines may provide the basis for a more general solution.

Our system depends on all requests receiving responses. As a result, replicas cannot be held accountable for refusing service and *Repeat and Compare* cannot prove a denial of service attack. We believe, however, that misbehaving nodes are motivated to serve rather than drop requests in order to maximize damage. As a result, denial of service is not a significant problem in CDNs. Another issue not addressed in this work is the scenario of an attacker strategically changing a small number of high-value responses to maximize damage. This issue is real because our probabilistic detection favors popular content. One possible solution is for origin servers to set a priority value in the attestation records of high-value content, with client forwarding such records with higher probability (e.g., $p = 1$).

Finally, our system does not provide support for databases. To generate attestation records for database reads, records would have to bind results to the corresponding queries and database tables. Supporting database updates, however, is more challenging. Since replicas are not trusted, our system can potentially suffer from forking attacks, where updates from different clients are hidden from each other [26]. We believe that database replication can provide the basis for a solution: different replicas can compare their local copies and thus detect divergence.

7 Conclusions

We have presented *Repeat and Compare*, a system for ensuring content integrity in peer-to-peer CDNs when replicas dynamically generate content. *Repeat and Compare* detects misbehaving replicas through attestation records and by leveraging the peer-to-peer network to repeat content generation on other replicas and then compare the results. Attestation records cryptographically bind replicas to their code, inputs, and dynamically generated output and build chains of accountability that help trace misbehavior. Our evaluation shows that *Repeat and Compare* is effective at quickly cleansing a CDN even if large fractions of replicas are misbehaving. 👾

Acknowledgments

We thank Mike Freedman, Guy Lichtman, Lakshmi Subramanian, Jinyang Li and Dinh Nguyen Tran for their discussions. We also thank our shepherd, Barbara Liskov, and the anonymous reviewers for their feedback. This material is based upon work supported by the National Science Foundation under Grant No. CNS-0537252.

References

[1] A. Adelsbach, M. Rohe, and A.-R. Sadeghi. Towards multilaterally secure digital rights distribution infrastructures. In *Proc. 5th ACM DRM*, pp. 45–54, Nov. 2005.

[2] A. S. Aiyer, L. Alvisi, A. Clement, M. Dahlin, J.-P. Martin, and C. Porth. BAR fault tolerance for cooperative services. In *Proc. 20th SOSP*, pp. 45–58, Oct. 2005.

[3] S. S. Bakken, A. Aulbach, E. Schmid, J. Winstead, L. T. Wilson, R. Lefdorf, A. Zmievski, and J. Ahto. *PHP Manual*. PHP Documentation Group, Feb. 2004. `http://www.php.net/manual/`.

[4] R. J. Bayardo and J. Sorensen. Merkle tree authentication of HTTP responses. In *Proc. 14th WWW*, pp. 1182–1183, May 2005.

[5] L. Bent, M. Rabinovich, G. M. Voelker, and Z. Xiao. Characterization of a large web site population with implications for content delivery. In *Proc. 13th WWW*, pp. 522–533, May 2004.

[6] C. Canali, V. Cardellini, M. Colajanni, R. Lancellotti, and P. S. Yu. Cooperative archictectures and algorithms for discovery and transcoding of multi-version content. In *Proc. 8th IWCW*, Sept. 2003.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM TOCS*, 20(4):398–461, Nov. 2002.

[8] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, Mar. 1996.

[9] C.-H. Chi and Y. Wu. An XML-based data integrity service model for web intermediaries. In *Proc. 7th IWCW*, 2002.

[10] B. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *Proc. 3rd NSDI*, pp. 45–58, May 2006.

[11] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. 20th SOSP*, pp. 133–147, Oct. 2005.

[12] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proc. 18th SOSP*, pp. 202–215, Oct. 2001.

[13] ECMA International. Common language infrastructure (CLI), 4th edition, June 2006.

[14] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI certificate theory. RFC 2693, IETF, Sept. 1999.

[15] M. J. Freedman. Personal Communication, Oct. 2006.

[16] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. 1st NSDI*, pp. 239–252, Mar. 2004.

[17] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *Proc. 3rd NSDI*, pp. 129–142, May 2006.

[18] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proc. 19th SOSP*, pp. 193–206, Oct. 2003.

[19] T. J. Giuli and M. Baker. Narses: A scalable, flow-based network simulator. Tech. Report arXiv:cs.PF/0211024, Stanford University, Nov. 2002.

[20] R. Grimm, G. Lichtman, N. Michalakis, A. Elliston, A. Kravetz, J. Miller, and S. Raza. Na Kika: Secure service execution and composition in an open edge-side computing network. In *Proc. 3rd NSDI*, pp. 169–182, May 2006.

[21] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The impact of DHT routing geometry on resilience and proximity. In *Proc. 2003 SIGCOMM*, pp. 381–394, Aug. 2003.

[22] I. Gupta, K. Birman, P. Linga, A. Demers, and R. van Renesse. Kelips: Building an efficient and stable P2P DHT through increased memory and background overhead. In *Proc. 2nd IPTPS*, pp. 160–169, Feb. 2003.

[23] A. Haeberlen, P. Kouznetsov, and P. Druschel. The case for byzantine fault detection. In *Proc. 2nd HotDep*, Nov. 2006.

[24] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. 21st PODC*, pp. 213–222, July 2002.

[25] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation management in P2P networks. In *Proc. 12th WWW*, pp. 640–651, May 2003.

[26] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (SUNDR). In *Proc. 6th OSDI*, pp. 121–136, Dec. 2004.

[27] J. Li, J. Stribling, R. Morris, and M. F. Kaashoek. Bandwidth-efficient management of DHT routing tables. In *Proc. 2nd NSDI*, pp. 99–114, May 2005.

[28] P. Maniatis, D. S. H. Rosenthal, M. Roussopoulos, M. Baker, T. J. Giuli, and Y. Muliadi. Preserving peer replicas by rate-limited sampled voting. In *Proc. 19th SOSP*, pp. 44–59, Oct. 2003.

[29] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. In *Proc. 1st Workshop on Internet Server Performance*, pp. 59–67, June 1998.

[30] OpenSSL. `http://www.openssl.org/`. Accessed Feb. 2007.

[31] H. K. Orman. Data integrity for mildly active content. *Proc. 3rd Workshop on Active Middleware Services*, p. 73, Aug. 2001.

[32] S. Pearson, B. Balacheff, L. Chen, D. Plaquin, and G. Proudler. *Trusted Computing Platforms: TCPA Technology In Context*. Prentice Hall, July 2002.

[33] Privoxy. `http://www.privoxy.org/`. Accessed Feb. 2007.

[34] B. Przydatek, D. Song, and A. Perrig. Sia: Secure information aggregation in sensor networks. In *Proc. 1st SenSys*, pp. 255–265, Nov. 2003.

[35] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou. Rx: Treating bugs as allergies—a safe method to survive software failures. In *Proc. 20th SOSP*, pp. 235–248, Oct. 2005.

[36] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. Middleware '01*, pp. 329–350, Nov. 2001.

[37] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM TOCS*, 2(4):277–288, Nov. 1984.

[38] A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla. Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems. In *Proc. 20th SOSP*, pp. 1–16, Oct. 2005.

[39] E. Shi, A. Perrig, and L. V. Doorn. BIND: A fine-grained attestation service for secure distributed systems. *Proc. 2005 S&P*, pp. 154–168, May 2005.

[40] W. Shi, K. Shah, Y. Mao, and V. Chaudhary. Tuxedo: A peer-to-peer caching system. In *Proc. 2003 PDPTA*, pp. 981–987, June 2003.

[41] A. Shieh, D. Williams, E. Sirer, and F. Schneider. Nexus: A new operating system for trustworthy computing. In *20th SOSP Work-in-Progress Session*, Oct. 2005.

[42] E. G. Sirer. Meridian: Data Description, 2005. `http://www.cs.cornell.edu/People/egs/meridian/data.php`. Accessed Feb. 2007.

[43] Y. J. Song, V. Ramasubramanian, and E. G. Sirer. Optimal resource utilization in content distribution networks. Tech. Report CIS TR2005-2004, Cornell University, Nov. 2005.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. 2001 SIGCOMM*, pp. 149–160, Aug. 2001.

[45] K. Walsh and E. G. Sirer. Experience with a distributed object reputation system for peer-to-peer filesharing. In *Proc. 3rd NSDI*, pp. 1–14, May 2006.

[46] L. Wang, V. Pai, and L. Peterson. The effectiveness of request redirection on CDN robustness. In *Proc. 5th OSDI*, pp. 345–360, Dec. 2002.

[47] Washington Post. The botnet trackers, February 16 2006. `http://www.washingtonpost.com/wp-dyn/content/article/2006/02/16/AR2006021601388.html`. Accessed Feb. 2007.

[48] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. 2005 SIGCOMM*, pp. 85–96, Aug. 2005.

[49] A. Yumerefendi and J. Chase. The Role of Accountability in Dependable Distributed Systems. In *Proc. 1st HotDep*, June 2005.

[50] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for network services. In *Proc. 11th ACM SIGOPS European Workshop*, p. 37, Sept. 2004.

[51] M. Yurkewych, B. N. Levine, and A. L. Rosenberg. On the cost-ineffectiveness of redundancy in commercial P2P computing. In *Proc. 12th CCS*, pp. 280–288, Nov. 2005.

[52] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A resilient global-scale overlay for service deployment. *IEEE J-SAC*, 22(1):41–53, Jan. 2004.

[53] W. Zhao and H. Schulzrinne. DotSlash: Providing dynamic scalability to web applications with on-demand distributed query result caching. Tech. Report CUCS-035-05, Columbia University, Sept. 2005.

[54] L. Zhou, F. B. Schneider, and R. V. Renesse. Coca: A secure distributed online certification authority. *ACM TOCS*, 20(4):329–368, Nov. 2002.

TightLip: Keeping Applications from Spilling the Beans

Aydan R. Yumerefendi, Benjamin Mickle, and Landon P. Cox
{aydan, bam11, lpcox}@cs.duke.edu
Duke University, Durham, NC

Abstract

Access control misconfigurations are widespread and can result in damaging breaches of confidentiality. This paper presents *TightLip*, a privacy management system that helps users define what data is sensitive and who is trusted to see it rather than forcing them to understand or predict how the interactions of their software packages can leak data.

The key mechanism used by TightLip to detect and prevent breaches is the *doppelganger* process. Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. The operating system runs a doppelganger and its original in parallel and uses divergent process outputs to detect potential privacy leaks.

Support for doppelgangers is compatible with legacy-code, requires minor modifications to existing operating systems, and imposes negligible overhead for common workloads. SpecWeb99 results show that Apache running on a TightLip prototype exhibits a 5% slowdown in request rate and response time compared to an unmodified server environment.

1 Introduction

Email, the web, and peer-to-peer file sharing have created countless opportunities for users to exchange data with each other. However, managing the permissions of the shared spaces that these applications create is challenging, even for highly skilled system administrators [15]. For untrained PC users, access control errors are routine and can lead to damaging privacy leaks. A 2003 usability study of the Kazaa peer-to-peer file-sharing network found that many users share their entire hard drive with the rest of the Internet, including email inboxes and credit card information [12]. Over 12 hours, the study found 156 distinct users who were sharing their email inboxes. Not only were these files available for download, but other users could be observed downloading them. Examples of similar leaks abound [16, 17, 22, 31].

Secure communication channels [3, 9] and intrusion detection systems [7, 11] would not have prevented these exposures. Furthermore, the impact of these leaks extends beyond the negligent users themselves since leaked sensitive data is often previous communication and transaction records involving others. No matter how careful any individual is, her privacy will only be as secure as her least competent confidant. Prior approaches to similar problems are either incompatible with legacy code [10, 14, 19, 26], rely on expensive binary rewriting and emulation [5, 20], or require changes to the underlying architecture [8, 28, 29].

We are exploring new approaches to preventing leaks due to access control misconfigurations through a privacy management system called *TightLip*. TightLip's goal is to allow organizations and users to better manage their shared spaces by helping them define *what* data is important and *who* is trusted, rather than requiring an understanding of the complex dynamics of *how* data flows among software components. Realizing this goal requires addressing three challenges: 1) creating file and host meta-data to identify sensitive files and trusted hosts, 2) tracking the propagation of sensitive data through a system and identifying potential leaks, and 3) developing policies for dealing with potential leaks. This paper focuses a new operating system object we have developed to deal with the second challenge: *doppelganger processes*.

Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. In TightLip, doppelgangers are spawned when a process tries to read sensitive data. The kernel returns sensitive data to the original and *scrubbed* data to the doppelganger. The doppelganger and original then run in parallel while the operating system monitors the sequence and arguments of their system calls. As long as the outputs for both processes are the same, then the original's output does not depend on the sensitive input with very high probability. However, if the operating system detects divergent outputs, then the original's output is likely descended from the sensitive input.

A breach arises when such an output is destined for an endpoint that falls outside of TightLip's control, such as a socket connected to an untrusted host. When potential breaches are detected, TightLip invokes a policy module, which can direct the operating system to fail the output, ignore the alert, or even swap in the doppelganger for the original. Using doppelgangers to infer the sensitivity of processes' outputs is attractive because it requires only minor changes to existing operating systems and no modifications to the underlying architecture or legacy applications.

We have added support for doppelgangers to the Linux kernel and currently support their use of the file system, UNIX domain sockets, pipes, network sockets, and GUIs. Early experience with this prototype has shown that doppelgangers are useful for an important subset of applications: servers which read files, encode the files' content, and then write the resulting data to the network. Micro-benchmarks of several common file transfer applications as well as the SpecWeb99 benchmark demonstrate that doppelgangers impose negligible performance overhead under moderate server workloads. For example, SpecWeb99 results show that Apache running on TightLip exhibits only a 5% slowdown in request rate and response time compared to an unmodified server environment.

2 Overview

Access control misconfigurations are common and potentially damaging: peer-to-peer users often inadvertently share emails and credit card information [12], computer science department faculty have been found to set the permissions of their email files to all-readable [31], professors have inadvertently left students' grade information in their public web space [22], a database of 20,000 Hong Kong police complainants' personal information was accidentally published on the web and ended up in Google's cache [16], and UK employees unintentionally copied sensitive internal documents to remote Google servers via Google Desktop [17]. Because these breaches were not the result of buggy or malicious software, they present a different threat model than is normally assumed by the privacy and security literature.

TightLip addresses this problem in three phases: 1) help users identify sensitive files, 2) track the propagation of sensitivity through a running system and detect when sensitive data may leave the system, and 3) enable policies for handling potential breaches. The focus of this paper is on the mechanisms used in phase two, but the rest of this section provides an overview of all three.

2.1 Identifying Sensitive Files

To identify sensitive data, TightLip periodically scans each file in a file system and applies a series of *diagnostics*, each corresponding to a different sensitive data type. These diagnostics use heuristics about a file's path, name, and content to infer whether or not it is of a particular sensitive type. For example, the email diagnostic checks for a ".pst" file extension, placement below a "mail" directory, and the ASCII string "Message-ID" in the file.

This scanning process is similar to anti-virus software that uses a periodically updated library of definitions to scan for infected files. The difference is that rather than prompting users when they find a positive match, diagnostics silently mark the file as sensitive and invoke the type's associated *scrubber*.

Scrubbers use a file's content to produce a non-sensitive *shadow* version of the file. For example, the email scrubber outputs a properly formatted shadow email file of the same size as the input file, but marks out each message's sender, recipient, subject, and body fields. Attachments are handled by recursively invoking other format-preserving, MIME-specific scrubbers. When the system cannot determine a data source's type it reverts to the default scrubber, which replaces each character from the sensitive data source with the "x" character.

2.2 Sensitivity Tracking and Breach Detection

Once files have been labeled, TightLip must track how sensitive information propagates through executing processes and prevent it from being copied to an untrusted destination. This problem is an instance of *information-flow* tracking, which has most commonly been used to protect systems from malicious exploits such as buffer overflows and format string attacks. Unfortunately, these solutions either suffer from incompatibly with legacy applications [10, 14, 19, 26, 32], require expensive binary rewriting [5, 6, 8, 20, 28], or rely on hardware support [29].

Instead, TightLip offers a new point in the design space of information-flow secure systems based on *doppelganger processes*. Doppelgangers are sandboxed copy processes that inherit most, but not all, of the state of an *original* process. Figure 1 shows a simple example of how doppelgangers can be used to track sensitive information. Initially, an original process runs without touching sensitive data. At some point, the original attempts to read a sensitive file, which prompts the TightLip kernel to spawn a doppelganger. The kernel returns the sensitive file's content to the original and the scrubbed content of the shadow file to the doppelganger.

Once the reads have been satisfied, the original and doppelganger are both placed on the CPU ready queue and, when scheduled, modify their private memory objects. The operating system subsequently tracks sensitivity at the granularity of a system call. If the doppelganger and original generate the same system call sequence with the same arguments, then these outputs do not depend on either the sensitive or scrubbed input with high probability and the operating system does nothing. This might happen when an application such as a virus scanner handles sensitive files, but does not act on their content.

However, if the doppelganger and original make the same system call with different arguments, then the original's output likely depends on sensitive data and the objects the call modifies are marked as sensitive. As long as updated objects are within the operating system's control, such as files and pipes, then they can be transitively

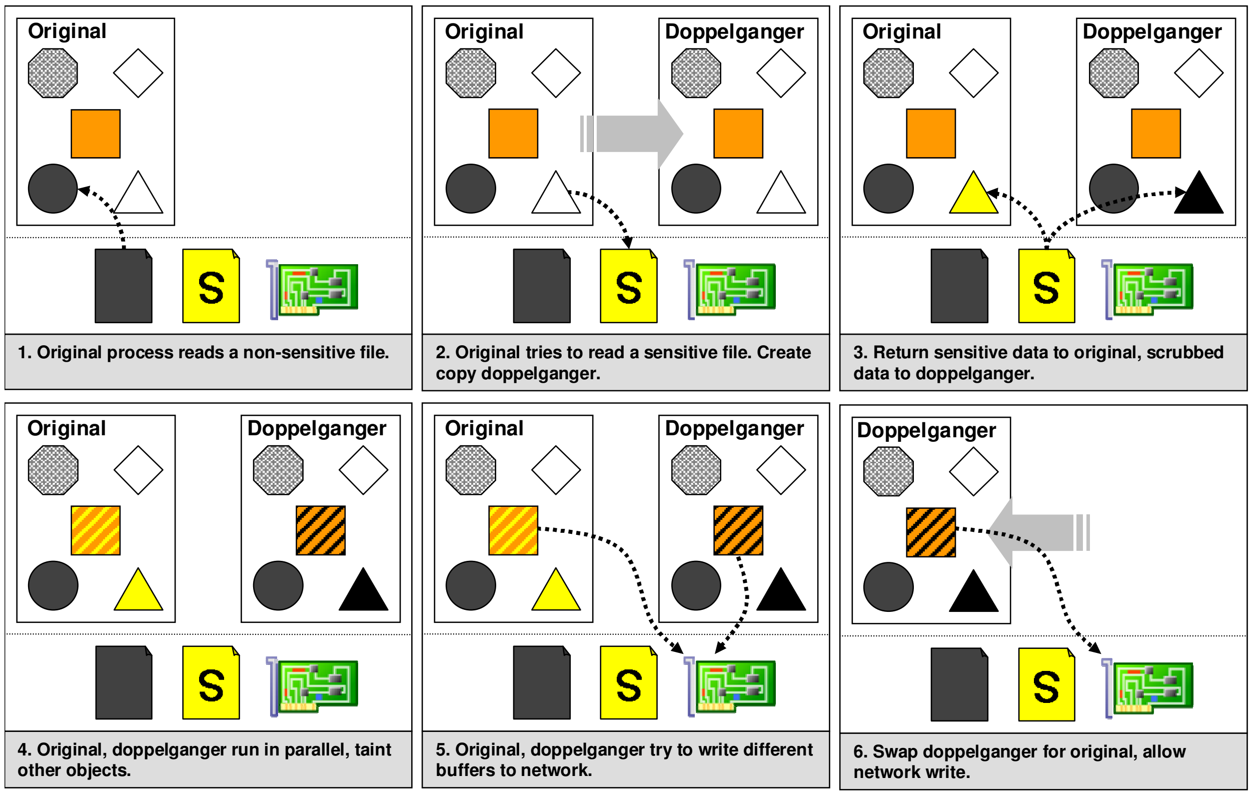

Original
Doppelganger
S
1. Original process reads a non-sensitive file.
2. Original tries to read a sensitive file. Create copy doppelganger.
3. Return sensitive data to original, scrubbed data to doppelganger.
4. Original, doppelganger run in parallel, taint other objects.
5. Original, doppelganger try to write different buffers to network.
6. Swap doppelganger for original, allow network write.

Figure 1: Using doppelgangers to avoid a breach.

labeled. However, if the system call modifies an object that is outside the control of the system, such as a socket connected to an untrusted host, then allowing the original's system call may compromise confidentiality.

By tracking information-flow at a relatively course granularity, TightLip avoids many of the drawbacks of previous approaches. First, because TightLip does not depend on any language-level mechanisms, it is compatible with legacy applications. Second, comparing the sequence and arguments of system calls does not require hardware support and needs only minor changes to existing operating systems. Third, the performance penalty of introducing doppelgangers is modest; the overhead of scheduling an additional process is negligible for most workloads.

Finally, an important limitation of existing information-flow tracking solutions is that they cannot gracefully transition a process from a tainted (i.e., having accessed sensitive data) and to an untainted state. A list of tainted memory locations or variables is not enough to infer what a clean alternative would look like. Bridging this semantic gap requires understanding a process's execution logic and data structure invariants. Because of this, once a breach has been detected, prior solutions require all tainted processes associated with the breach to be rebooted. While rebooting purges taint, it also wipes out untainted connections and data structures.

Doppelgangers provide TightLip with an internally consistent, clean alternative to the tainted process; as long as shadow files are generated properly, doppelgangers will not contain any sensitive information. This allows TightLip to swap doppelgangers in for their original processes—preserving continuous execution without compromising confidentiality.

2.3 Disclosure Policies

Once the operating system detects that sensitive data is about to be copied onto a destination outside of TightLip's control, it invokes the *disclosure policy module*. Module policies specify how the kernel should handle attempts to copy sensitive data to untrusted destinations. Our current prototype supports actions such as disabling a process's write permissions, terminating the process, scrubbing output buffers, or swapping the doppelganger in for the original.

TightLip provides default policies, but also notifies users of potential breaches so that they can define their own policies. Query answers can be delivered synchronously or asynchronously (e.g. via pop-up windows or emails). Answers can also be cached to minimize future interactions with the user.

3 Limitations

Though TightLip is attractive for its low overhead, compatibility with legacy applications and hardware, and support for continuous execution, it is not without its limitations. First, in TightLip the operating system is completely trusted. TightLip is helpless to stop the kernel from maliciously or unintentionally compromising confidentiality. For example, TightLip cannot prevent an in-kernel NFS server from leaking sensitive data.

Second, TightLip relies on scrubbers to produce valid data. An incorrectly formatted shadow file could crash the doppelganger. In addition, swapping in a doppelganger is only safe if scrubbers can remove all sensitive information. While feasible for many data types, it may not be possible to meet these requirements for all data sources.

Third, scrubbed data can lead to false negatives in some pathological cases. For example, an original process may accept a network query asking whether a sensitive variable is even or odd. TightLip could generate a scrubbed value that is different from the sensitive variable, but of the same parity. The output for the doppelganger and original would be the same, despite the fact that the original is leaking information. The problem is that it is possible to generate "unlucky" scrubbed data that can lead to a false negative. Such false negatives are unlikely to arise in practice since the probability of a collision decreases geometrically with the number of bits required to encode a response.

Fourth, TightLip avoids the overhead of previous approaches by focusing on system calls, rather than individual memory locations. Unfortunately, if a process reads sensitive data from multiple sources, TightLip cannot compute the exact provenance of a sensitive output. While this loss of information makes more fine-grained confidentiality policies impossible, it allows us to provide practical application performance.

Fifth, TightLip does not address the problem of covert channels. An application can use a variety of covert channels to transmit sensitive information [24, 29]. Since it is unlikely that systems can close all possible covert channels [29], dealing with covert channels is beyond the scope of this paper.

Finally, TightLip relies on comparisons of process outputs to track sensitivity and transitively label objects. If the doppelganger generates a different system call than its original, it has entered a different execution state and may no longer provide information about the relationship between the sensitive input and the original's output. Such divergence might happen if scrubbed input induced a different control flow in the doppelganger. Without the doppelganger as a point of comparison, any object subsequently modified by the original must be marked sensitive; this can lead to incorrectly labeled objects and false positives.

This limitation of doppelgangers is similar to those faced by taint-flow analysis of "implicit flow." Consider the following code fragment, in which variable `x` is tainted: `if(x) { y=1; } else { y=0; }`. Variable `y` should be flagged since its value depends on the value of `x`. Tainting each variable written inside a conditional block captures all dependencies, but can also implicate innocent variables and raise false positives. In practice, following dependencies across conditionals is extremely difficult without carefully-placed programmer annotations [32]. Every taint-checker for legacy code that we are aware of ignores implicit flow to avoid false positives.

Despite the challenges of conditionals, for an important subset of applications, it is reasonable to assume that scrubbed input will not affect control flow. Web servers, peer-to-peer clients, distributed file systems, and the sharing features of Google Desktop blindly copy data into buffers without interpreting it. Early experience with our prototype confirms such behavior and the rest of this paper is focused on scenarios in which scrubbed data does not affect control flow.

Much of the rest of our discussion of TightLip describes how to eliminate sources of divergence between an original and doppelganger process so that differences only emerge from the initial scrubbed input. If any other input or interaction with the system causes a doppelganger to enter an alternate execution state, TightLip may generate additional false positives.

4 Design

There are two primary challenges in designing support for doppelganger processes. First, because doppelgangers may run for extended periods and compete with other processes for CPU time and physical memory, they must be as resource-efficient as possible. Second, since TightLip relies on divergence to detect breaches, all doppelganger inputs and outputs must be carefully regulated to minimize false positives.

4.1 Reducing Doppelganger Overhead

Our first challenge was limiting the resources consumed by doppelgangers. A doppelganger can be spawned at any point in the original's execution. One option is to create the doppelganger concurrently with the original, but doing so would incur the cost of monitoring in the common case when taint is absent.

Instead, TightLip only creates a doppelganger when a process attempts to read from a sensitive file. For the vast majority of processes, reading sensitive files will occur rarely, if ever. However, some long-lived processes that frequently handle sensitive data such as virus scanners and file search tools may require a doppelganger

Type	Example	Processing description
Kernel update	bind	Apply original, return result to both.
Kernel read	getuid	Verify identical system call sequences.
Non-kernel update	send	Synchronize, compare buffers.
Non-kernel read	gettimeofday	Buffer original results, return to both.

Table 1: Doppelganger-kernel interactions.

throughout their execution. For these applications, it is important that doppelgangers be as resource-efficient as possible.

Once created, doppelgangers are inserted into the same CPU ready queue as other processes. This imposes a modest scheduling overhead and adds processor load. However, unlike taint-checkers, the fact that doppelgangers have a separate execution context enables a degree of parallelization with other processes, including the original. Though we assume a uni-processor environment throughout this paper, TightLip should be able to take advantage of emerging multi-core architectures.

To limit memory consumption, doppelgangers are forked from the original with their memory marked copy-on-write. In addition, nearly all of the doppelganger's kernel-maintained process state is shared read-only with the original, including its file object table and associated file descriptor namespace. The only separate, writable objects maintained for the doppelganger are its execution context, file offsets, and modified memory pages.

4.2 Doppelganger Inputs and Outputs

In TightLip the kernel must manage doppelganger inputs and outputs to perform three functions: prevent external effects, limit the sources of divergence to the initial scrubbed input, and contain sensitive data. To perform these functions, the kernel must regulate information that passes between the doppelganger and kernel through system calls, signals, and thread schedules.

Kernel-doppelganger interactions fall into one of the following categories: *kernel updates*, *kernel reads*, *non-kernel updates*, and *non-kernel reads*. Table 1 lists each type, provides an example system call, and briefly describes how TightLip regulates the interaction.

4.2.1 Updates to Kernel State

As with speculative execution environments [4, 21], TightLip must prevent doppelgangers from producing any external effects so that it remains unintrusive. As long as an application does not try to leak sensitive information, it should behave no differently than in the case when there is no doppelganger.

This is why original processes must share their kernel state with the doppelganger read-only. If the doppelganger were allowed to update the original's objects, it could alter its execution. Thus, system calls that modify the shared kernel state must be strictly ordered so that only the original process can apply updates.

System calls that update kernel state include, but are not limited to, exit, fork, time, lseek, alarm, sigaction, gettimeofday, settimeofday, select, poll, llseek, fcntl, bind, connect, listen, accept, shutdown, and setsockopt.

TightLip uses barriers and condition variables to implement these system calls. A barrier is placed at the entry of each kernel modifying call. After both processes have entered, TightLip checks their call arguments to verify that they are the same. If the arguments match, then the original process executes the update, while the doppelganger waits. Once the original finishes, TightLip notifies the doppelganger of the result before allowing it to continue executing.

If the processes generate different updates and the modified objects are under the kernel's control, TightLip applies the original's update and records a transfer of sensitivity. For example, the kernel transitively marks as sensitive objects such as pipes, UNIX domain sockets, and files. Subsequent reads of these objects by other processes may spawn doppelgangers.

It is important to note that processes will never block indefinitely. If one process times out waiting for the other to reach the barrier, TightLip assumes that the processes have diverged and discards the doppelganger. The kernel will then have to either mark any subsequently modified objects sensitive or invoke the policy module.

Signals are a special kernel-doppelganger interaction since they involve two phases: signal handler registration, which modifies kernel data, and signal delivery, which injects data into the process. Handler registration is managed using barriers and condition variables as other kernel state updates are; only requests from the original are actually registered. However, whenever signals are delivered, both processes must receive the same signals in the same order at the same points in their execution. We discuss signal delivery in Section 4.2.2.

Of course, doppelgangers must also be prevented from modifying non-kernel state such as writing to files or network sockets. Because it may not be possible to proceed with these writes without invoking a disclosure policy and potentially involving the user, modifications of non-kernel state are treated differently. We discuss updates to non-kernel state in Section 4.2.3.

4.2.2 Doppelganger Inputs

To reduce the false positive rate TightLip must ensure that sources of divergence are limited to the scrubbed input. For example, both processes must receive the same values for time-of-day requests, receive the same network data, and experience the same signals. Ensuring that reads from kernel state are the same is trivial, given that updates are synchronized. However, preventing non-kernel reads, signal delivery, and thread-interleavings from generating divergence is more challenging.

Non-kernel reads

The values returned by non-kernel reads, such as from a file, a network socket, and the processor's clock, can change over time. For example, consecutive calls to gettimeofday or consecutive reads from a socket will each return different data. TightLip must ensure that paired accesses to non-kernel state return the same value to both the original and doppelganger. This requirement is similar to the *Environment Instruction Assumption* ensured by hypervisor-based fault-tolerance [2].

To prevent the original from getting one input and the doppelganger another, TightLip assigns a producer-consumer buffer to each data source. For each buffer, the original process is the producer and the doppelganger is the consumer. System calls that use such queues include read, readv, recv, recvfrom, and gettimeofday.

If the original (producer) makes a read request first, it is satisfied by the external source and the result is copied into the buffer. If the buffer is full, the original must block until the doppelganger (consumer) performs a read from the same non-kernel source and consumes the same amount of data from the buffer. Similarly, if the doppelganger attempts to read from a non-kernel source and the buffer is empty, it must wait for the original to add data.

The mechanism is altered slightly if the read is from another sensitive source. In this case, the kernel returns scrubbed buffers to the doppelganger and updates a list of sensitive inputs to the process. Otherwise, the producer-consumer queue is handled exactly the same as for a non-sensitive source. As before, neither process will block indefinitely.

Signals

In Section 4.2.1, we explained that signals are a two-phase interaction: a process registers a handler and the kernel may later deliver a signal. We treat the first phase as a kernel update. Since modifications to kernel state are synchronized, any signal handler that the original successfully registers is also registered for the doppelganger.

The TightLip kernel delivers signals to a process as it transitions into user mode. Any signals intended for a process are added to its signal queue and then moved from the queue to the process's stack as it exits kernel space. A process can exit kernel space either because it has finished a system call or because it had been preempted and is scheduled to start executing again.

To prevent divergence, any signals delivered to the doppelganger and original must have the same content, be delivered in the same order, and must be delivered to the same point in their execution. If any of these conditions are violated, the processes could stray. TightLip ensures that signal content and order is identical by copying any signal intended for the original to both the original's and doppelganger's signal queue.

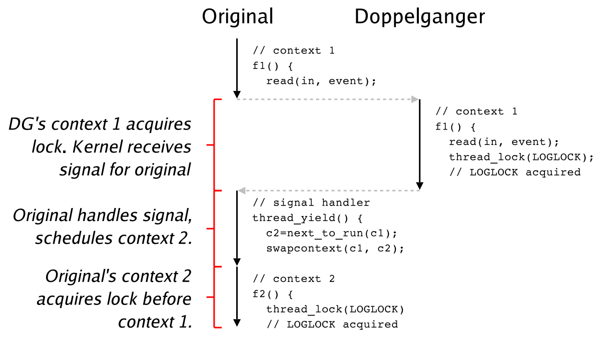
Before jumping back into user space, the kernel places pending signals on the first process's stack. Conceptually, when the process re-enters user space, it handles the signals in order before returning from its system call. The same is true when the second process (whether the doppelganger or original) re-enters user space. For the second process, a further check is needed to ensure that only signals that were delivered to the first are delivered to the second.

In previous sections, we have described how the original and doppelganger must be synchronized when entering system call code in the kernel so that TightLip can detect divergence. Unfortunately, simply synchronizing the entry to system calls between processes is insufficient to ensure that signals are delivered to the same execution state.

This is because some system calls can be interrupted by a signal arriving while the kernel is blocked waiting for an external event to complete the call. In such cases, the kernel delivers the signal to the process and returns an "interrupted" error code (e.g. EINTR in Linux). Interrupting the system call allows the kernel to deliver signals without waiting (potentially forever) for the external event to occur.

Properly written user code that receives an interrupted error code will retry the system call. If TightLip only synchronizes on system call entry-points, retrying an interrupted system call can lead to different system call sequences. Consider the following example taken from the execution of the SSH daemon, sshd, where Process 1 and 2 could be either the doppelganger or original:

- Process 1 (P1) calls write and waits for Process 2 (P2).
- P2 calls write, wakes up P1, completes write, returns to user-mode, calls select, and waits for P1 to call select.
- P1 wakes up and begins to complete write.
- A signal arrives for the original process.
- The kernel puts the signal handler on P1's stack and sets the return value of P1's write to EINTR.


Original
Doppelganger
DG's context 1 acquires lock. Kernel receives signal for original
Original handles signal, schedules context 2.
Original's context 2 acquires lock before context 1.

```
// context 1
f1() {
  read(in, event);
```

```
// context 1
f1() {
  read(in, event);
  thread_lock(LOGLOCK);
  // LOGLOCK acquired
```

```
// signal handler
thread_yield() {
  c2=next_to_run(c1);
  swapcontext(c1, c2);
```

```
// context 2
f2() {
  thread_lock(LOGLOCK)
  // LOGLOCK acquired
```

Figure 2: Signaling that leads to divergence.

- P1 handles the signal, sees a return code of EINTR for write, retries write, and waits for P2 to call write.

In this example, divergence arose because P1's and P2's calls to write generated different return values, which led P1 to call write twice. To prevent this, TightLip must ensure that paired system calls generate the same return values. Thus, system call exit-points must be synchronized as well as entry-points. In our example, paired exit-points prevent P2's write from returning a different return value than P1's: both P1 and P2 are returned either EINTR or the number of written bytes.

Parallel control flows as well as lock-step system call entry and exit points make it likely that signals will be delivered to the same point in processes' execution, but they are still not a guarantee. To see why, consider the processes in Figure 2. In the example, a user-level thread library uses an alarm signal to pre-empt an application's threads. When the signal is handled determines how much progress the user-level thread makes. In this case, it determines the order in which threads acquire a lock. The problem is that the doppelganger and original have been pre-empted at different instructions, which forces them to handle the same signal in different states. Ideally, the processor would provide a *recovery register*, which can be decremented each time an instruction is retired; the processor then generates an interrupt once it becomes negative. Unfortunately, the x86 architecture does not support such a register.

Even without a recovery register, TightLip can still limit the likelihood of divergence by deferring signal delivery until the processes reach a synchronization point. Most programs make system calls throughout their execution, providing many opportunities to handle signals. However, for the rare program that does not make any system calls, the kernel cannot wait indefinitely without compromising program correctness. Thus, the kernel can defer delivering signals on pre-emption re-entry only a finite number of times. In our limited experience with our prototype kernel, we have not seen process divergence due to signal delivery.

Threads

Managing multi-threaded processes requires two additional mechanisms. First, the kernel must pair doppelganger and original threads entering and exiting the kernel. Second, the kernel must ensure that synchronization resources are acquired in the same order for both processes. Assuming parallel control flows, if control is transferred between threads along system calls and thread primitives such as lock/unlock pairs, then TightLip can guarantee that the original and doppelganger threads will enter and exit the kernel at the same points.

4.2.3 Updates to Non-kernel State

The last process interactions to be regulated are updates to non-kernel state. As with other system calls, these updates are synchronized between the processes using barriers and condition variables. The difference between these modifications and those to kernel state is that TightLip does not automatically apply the original's update and return the result to both processes. TightLip's behavior depends on whether the original and the doppelganger have generated the same updates.

Handling Potential Leaks

If both processes generate the same update, then TightLip assumes that the update does not depend on the sensitive input and that releasing it will not compromise confidentiality. The kernel applies the update, returns the result, and takes no further action.

If the updates differ and are to an object outside of the kernel's control, TightLip assumes that a breach is about to occur and queries the disclosure policy module. Our prototype currently supports several disclosure policies: do nothing (allow the potentially sensitive data to pass), disable writes to the network (the system call returns an error), send the doppelganger output instead of the original's, terminate the process, and swap the doppelganger for the original process.

Swapping

If the user chooses to swap in the doppelganger, the kernel sets the original's child processes' parent to the doppelganger, discards the original, and associates the original's process identifier with the doppelganger's process state. While the swap is in-progress, both processes must be removed from the CPU ready queue. This allows related helper processes to make more progress than they might have otherwise, which can affect the execution of the swapped-in process in subtle but not incorrect ways. We will describe an example of such behavior in Section 6.1.

Swapped-in processes require an extra mechanism to run the doppelganger efficiently and safely. For each swapped-in process, TightLip maintains a fixed-size list of open files inherited from the doppelganger. Anytime the swapped-in process attempts to read from a sensitive file, the kernel checks whether the file is on the list. If it is, TightLip knows that the process had previously received scrubbed data from the file and returns more scrubbed data. If the file is not on the list and the file is sensitive, TightLip spawns a new doppelganger.

These lists are an optimization to avoid spawning doppelgangers unnecessarily. Particularly for large files that require multiple reads, spawning a new doppelganger for every sensitive read can lead to poor performance. Importantly, leaving files off of the list can only hurt performance and will never affect correctness or compromise confidentiality. Because of this guarantee, TightLip can remove any write restrictions on the swapped-in process since its internal state is guaranteed to be untainted.

Unfortunately, swapping is not without risk. In some cases, writing the doppelganger's buffer to the network and keeping the doppelganger around to monitor the original may be the best option. For example, the user may want the original to write sensitive data to a local file even if it should not write it to the network. However, maintaining both processes incurs some overhead and non-sensitive writes would still be identical for both the original and the swapped-in process with very high probability.

Furthermore, the doppelganger can stray from the original in unpredictable ways. This is similar to the uncertainty generated by failure-oblivious computing [23]. To reduce this risk, TightLip can monitor internal divergence in addition to external divergence. External symptoms of straying are obvious—when the doppelganger generates different sequences of system calls or uses different arguments. Less obvious may be if the scrubbed data or some other input silently shifts the process's control flow. Straying of this form may not generate external symptoms, but can still leave the doppelganger in a different execution state than the original.

We believe that this kind of divergence will be rare for applications such as file servers, web servers, and peer-to-peer clients; these processes will read a sensitive file, encode its contents, and write the result to the network. Afterward, the doppelganger and original will return to the same state after the network write. In other cases, divergence will likely manifest itself as a different sequence of system calls or a crash [18].

For additional safety, TightLip can take advantage of common processor performance counters, such as those offered by the Pentium4 [27] to detect internal divergence. If the number of instructions, number of branches taken and mix of loads and stores are sufficiently similar, then it is unlikely that the scrubbed input affected the doppelganger's control flow. TightLip can use these values to measure the likelihood that the doppelganger and original are in the same execution state and relay this information to the user.

4.3 Example: Secure Copy (scp)

To demonstrate the design of the TightLip kernel, it is useful to step through an example of copying a sensitive file from a TightLip-enabled remote host via the secure copy utility, scp.

Secure copy requests are accepted by an SSH daemon, sshd, running on the remote host. After authenticating the requester, sshd forks a child process, shell, which runs under the uid of the authenticated user and will transfer encrypted file data directly to the requester via a network socket, nsock. shell creates a child process of its own, worker, which reads the requested data from the file system and writes it to a UNIX domain socket, dsock, connecting shell and worker.

As soon as worker attempts to read a sensitive file, the kernel spawns a doppelganger, D(worker). Once worker and D(worker) have returned from their respective reads, they both try to write to dsock. Since dsock is under the kernel's control, the actual file data from worker is buffered and dsock is transitively marked sensitive. shell, meanwhile, selects on dsock and is woken up when there is data available for reading.

When shell attempts to read from dsock (which is now sensitive), the kernel forks another doppelganger, D(shell), and returns the actual buffer content (sensitive file data) to shell and scrubbed data to D(shell). shell and D(shell) both encrypt their data and attempt to write the result to nsock. Since their output buffers are different, the breach is detected. By default, the kernel writes D(shell)'s encrypted scrubbed data to nsock, sets the parent process of worker and D(worker) to D(shell), and swaps in D(shell) for shell.

4.4 Future Work

Though TightLip supports most interactions between doppelgangers and the operating system, there is still some work to be done. For example, we currently do not support communication over shared memory. TightLip could interpose on individual loads and stores to shared memory by setting the page permissions to read-only. Though this prevents sensitive data from passing freely, it also generate a page fault on every access of the shared pages.

In addition, TightLip currently lacks a mechanism to prevent a misconfigured process from overwriting sensitive data. Our design targets data confidentiality, but does not address data integrity. However, it is easy to imagine integrating integrity checks with our current design. For example, anytime a process attempts to write to a sensitive file, TightLip could invoke the policy module, as it currently does for network socket writes.

Finally, we believe that it will be possible to reduce the memory consumed by a long-lived doppelganger by periodically comparing its memory pages to the original's. This would make using the doppelganger solely to generate untainted network traffic—as opposed to swapping it in for the original—more attractive.

Though doppelgangers will copy-on-write memory pages as they execute, many of those pages may still be identical to the original's. This would be true for pages that only receive updates that are independent of the scrubbed input. These pages could be remarked copy-on-write and shared anew by the two processes.

Furthermore, even if a page initially contained bytes that depended on the scrubbed input, over time those bytes may be overwritten with non-sensitive values. These pages could also be recovered. Carried to its logical conclusion, if all memory pages of the original and doppelganger converged, then the doppelganger could be discarded altogether. We may be able to apply the memory consolidation techniques used in the VMware hypervisor [30] to this problem and intend to explore these mechanisms and others in our future work.

5 Implementation

Our TightLip prototype consists of several hundred lines of C code scattered throughout the Linux 2.6.13 kernel. We currently support signals, inter-process communication via pipes, UNIX domain sockets, and graphical user interfaces. Most of the code deals with monitoring doppelganger execution, but we also made minor modifications to the `ext3` file system to store persistent sensitivity labels.

5.1 File Systems

Sensitivity is currently represented as a single bit co-located on-disk with file objects. If more complex classifications become necessary, using one bit could be extended to multiple bits. To query sensitivity, we added a predicate to the kernel file object that returns the sensitivity status of any file, socket, and pipe. TightLip currently only supports sensitivity in the `ext3` file system, though this implementation is backwards-compatible with existing `ext3` partitions. Adding sensitivity to future file systems should be straightforward since manipulating the sensitivity bit in on-disk `ext3` inodes only required an extra three lines of code.

Our prototype also provides a new privileged system call to manage sensitivity from user-space. The system call can be used to read, set, or clear the sensitivity of a given file. This is used by TightLip diagnostics and by a utility for setting sensitivity by hand.

5.2 Data Structures

Our prototype augments several existing Linux data structures and adds one new one, called a *completion structure*. Completion structures buffer the results of an invoked kernel function. This allows TightLip to apply an update or receive a value from a non-kernel source once, but pass on the result to both the original and doppelganger. Minimally, completion structures consist of arguments to a function and its return value. They may also contain instructions for the receiving process, such as a divergence notification or instructions to terminate.

TightLip also required several modifications to the Linux task structure. These additions allow the kernel to map doppelgangers to and from originals, synchronize their actions, and pass messages between them. The task structure of the original process also stores a list of buffers corresponding to kernel function calls such as bind, accept, and read. Finally, all process structures contain a list of at most 10 open sensitive files from which scrubbed data should be returned. Once a sensitive file is closed, it is removed from this list.

5.3 System Calls

System call entry and exit barriers are crucial for detecting and preventing divergence. For example, correctly implementing the exit system call requires that peers synchronize in the kernel to atomically remove any mutual dependencies between them. We have inserted barriers in almost all implemented system calls. In the future, we may be able to relax these constraints and eliminate some unnecessary barriers.

We began implementing TightLip by modifying read system calls for files and network sockets. Next, we modified the write system call to compare the outputs of the original and the doppelganger. The prototype allows invocation of a custom policy module when TightLip determines that a process is attempting to write sensitive data. Supported policies include allowing the sensitive data to be written, killing the process, closing the file/socket, writing the output of the doppelganger, and swapping the doppelganger for the original process.

After read and write calls, we added support for reads and modifications of kernel state, including all of the socket system calls. We have instrumented most, but not all relevant system calls. Linux currently offers more than 290 system calls, of which we have modified 28.

5.4 Process Swapping

TightLip implements process swapping in several stages. First, it synchronizes the processes using a barrier. Then the original process notifies the doppelganger that swapping should take place. The doppelganger receives the message and exchanges its process identifier with the original's. To do this requires unregistering both processes from the global process table and then re-registering them under the exchanged identifiers. The doppelganger must then purge any pointers to the original process's task structure.

Once the doppelganger has finished cleaning up, it acknowledges the original's notification. After receiving this acknowledgment, the original removes any of its state that depends on the doppelganger and sets its parent to the `init` process. This avoids a child death signal from being delivered to its actual parent. The original also re-parents all of its children to the swapped-in doppelganger. Once these updates are in place, the original safely exits.

5.5 Future Implementation Work

There are still several features of our design that remain unimplemented. The major goal of the current prototype has been to evaluate our design by running several key applications such as a web server, NFS server, and sshd server. We are currently working on support for multi-threaded applications. Our focus on single-threaded applications, pipes, UNIX domain sockets, files, and network sockets has given us valuable experience with many of the core mechanisms of TightLip and we look forward to a complete environment in the very near future.

6 Evaluation

In this section we describe an evaluation of our TightLip prototype using a set of data transfer micro-benchmarks and SpecWeb99. Our goal was to examine how TightLip affects data transfer time, resource requirements, and application saturation throughput.

We used several unmodified server applications: Apache-1.3.34, NFS server 2.2beta47-20, and sshd-3.8. Each of these applications is structured differently, leading to unique interactions with the kernel. Apache runs as a collective of worker processes that are created on demand and destroyed when idle for a given period. The NFS server is a single-threaded, event-driven process that uses signals to handle concurrent requests.

sshd forks a shell process to represent the user requesting a connection. The shell process serves data transfer requests by forking a worker process to fetch files from the file system. The worker sends the data to the shell process using a UNIX domain socket, and the shell process encrypts the data and sends it over the network to the client. All sshd forked processes belonging to the same session are destroyed when the client closes the connection.

All experiments ran on a Dell Precision 8300 workstation with a single 3.0 GHz Pentium IV processor and 1GB RAM. We ran all client applications on an identical machine connected to the TightLip host via a closed 100Mbs LAN. All graphs report averages together with a 95% confidence interval obtained from 10 runs of each experiment. It should be noted that we did not detect any divergence prior to the network write for any applications during our experiments.

6.1 Application Micro-benchmarks

In this set of experiments we examined TightLip's impact on several data transfer applications. We chose these applications because they are typical of those likely to inadvertently leak data, as exemplified by the motivating Kazaa, web server, and distributed file system misconfigurations [12, 16, 22, 31]. Our methodology was simple; each experiment consisted of a single client making 100 consecutive requests for 100 different files, all of the same size. As soon as one request finished, the client immediately made another.

For each trial, we examined four TightLip configurations. To capture baseline performance, each server initially ran with *no sensitive files*. The server simply read from the file system, encoded the files' contents, and returned the results over the network.

Next, we ran the servers with all files marked sensitive and applied three more policies. The *continuous* policy created a doppelganger for each process that read sensitive data and ran the doppelganger alongside the original until the original exited. Subsequent requests to the original process were also processed by the doppelganger.

The *swap* policy followed the continuous policy, but swapped in the doppelganger for the original after each network write. If the swapped-in process accessed sensitive data again, a new doppelganger was created and swapped in after the next write.

The *optimized swap* policy remembered if a process had been swapped in. This allowed TightLip to avoid creating doppelgangers when the swapped process attempted to further read from the same sensitive source; the system could return scrubbed data without creating a new doppelganger.

Figure 3, Figure 4, and Figure 5 show the relative transfer times for the above applications when clients fetched sensitive files of varying sizes.

Note that the cost of the additional context switches TightLip requires to synchronize the original and doppelganger may be high relative to the baseline transfer time for smaller files. This phenomenon is most noticeable for the NFS server in Figure 4, where fetching files of size 1K and 4K was 30% and 25% more expensive, respectively, than fetching non-sensitive files. As file size increases, data transfer began to dominate the context switch overhead induced by TightLip; the NFS server running under all policies transferred 256KB within 10% of the baseline time.

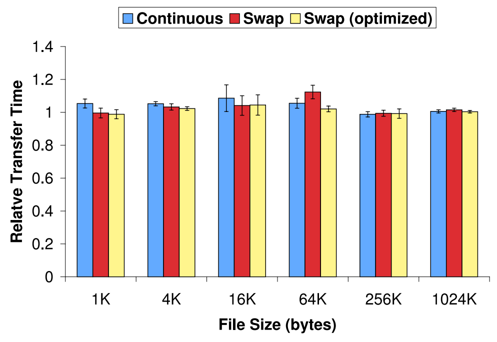


Figure 3: Apache relative transfer time.

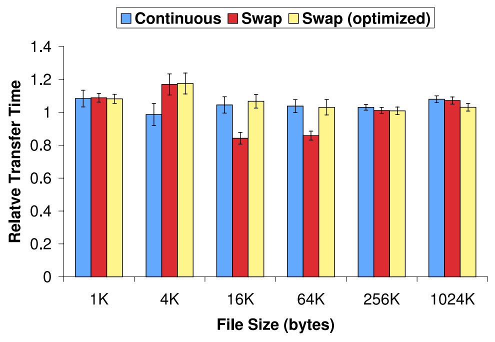


Figure 5: SSH relative transfer time.

Server	Continuous	Swap	Optimized
apache	852	76634	5389
sshd	76277	166055	38085
nfsd	58	233017	42395

Table 2: Average total number of additional pages created during a run of the data transfer micro-benchmarks. Each run transfers 600 files for a total of 133MB.

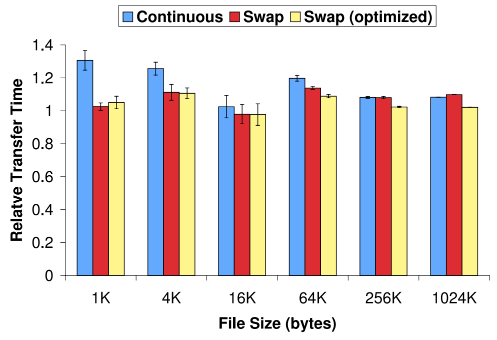


Figure 4: NFS relative transfer time.

Figure 3 shows that our Apache web server was the least affected by the TightLip. The overhead under all three policies was within 5% of the baseline, with continuous execution being slightly more expensive than the other two. This result can be explained by the fact that fetching static files from the web server was I/O bound and required little CPU time. Continuous execution was slightly more expensive, since the original and the doppelganger both parsed every client request.

Figure 5 shows that the overhead of using doppelgangers for sshd was within 10% of the baseline for most cases. This was initially surprising, since the original and doppelganger performed encryption on the output concurrently. However, the overhead of performing extra symmetric encryption was low and masked by the more dominant cost of I/O.

The swap policy performed better than the continuous execution policy for Apache and NFS. This result was expected since process swapping reduces the overhead of running a doppelganger. The benefit from process swapping was application-dependent though, as the time spent swapping the doppelganger for the original sometimes outweighed the overhead incurred by running the doppelganger—while swapping took place, the process was blocked and could not make any progress. Transferring 4K size files from sshd illustrated this point: sshd was almost done transferring all of its data after the first write to the network. Swapping the doppelganger for the original only delayed completion of the request.

To our surprise, the swapping policy applied to sshd actually reduced transfer times for 16K and 64K files. The reason for this behavior was that during swapping, the sshd shell process blocked and could not consume data from the UNIX domain socket. However, the worker process continued to feed data to the socket, which increased the amount of data the shell process found on its next read.

Since the shell process had a larger read buffer than the worker process, swapping caused the shell process to perform larger reads and, as a result, fewer network writes relative to not swapping. Performing fewer system calls improved the transfer time observed by the client. The impact of swapping decreased as file size increased since the fixed-size buffer of the UNIX domain socket forced worker processes to block if the socket was full.

The optimized swap policy had the best overall performance among all three policies. Since all servers perform repeated reads from the same sensitive source, creating doppelgangers after every read was unnecessary. Even though this policy often improved performance, it did not apply in all cases. The policy assumed that sensitive writes depended on *all* sensitive sources that a process had opened. Thus, future reads from these sensitive sources always produced scrubbed data.

Doppelgangers affected memory usage as well as response time. Table 2 shows the average total number of extra memory pages allocated while running the entire benchmark. Each cell represents the additional number of pages created during the transfer of all 600 files (133MB).

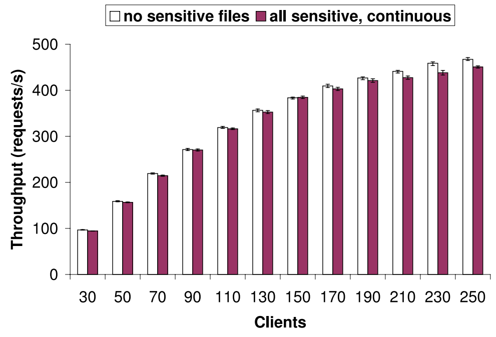


Figure 6: SpecWeb99 throughput.

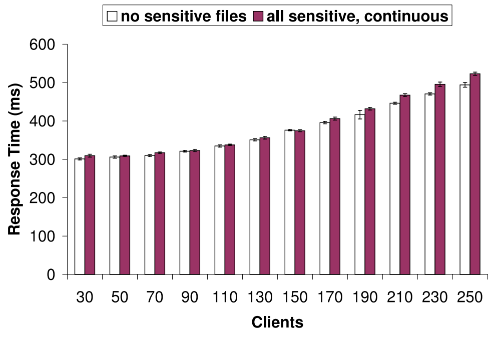


Figure 7: SpecWeb99 response time.

We observed that the server applications behave differently under our policies. The best memory policy for Apache and nfsd was continuous execution since for both servers' process executes until the end of the benchmark. For these two servers any other policy increased the number of doppelgangers created and required more page allocations. Since an sshd process only executes for the duration of a single file transfer, continuous execution was not as good as swap-optimized execution. For all three servers, the swap policy produced the most page allocations, since it created more doppelgangers.

Overall, our micro-benchmark results suggest that TightLip has low impact on data transfer applications. The overhead depends on the policy used to deal with sensitive writes. In most cases the overhead was within 5%, and it never exceeded 30%. Even with doppelgangers running continuously, TightLip outperformed prior taint-checking approaches by many orders of magnitude. For example, Apache running under TaintCheck and serving 10KB files is nearly 15 times slower than an unmodified server. For 1KB files, it is 25 times slower [20]. Thus, even in the worst case, using doppelgangers provides a significant performance improvement for data transfer applications.

6.2 Web Server Performance

Our final set of experiments used the SpecWeb99 benchmark on an Apache web server running on a TightLip machine. We used two configurations for these experiments—no sensitive files and continuous execution with all files marked sensitive. Since the benchmark verified the integrity of every file, we configured TightLip to return the data supplied by the original instead of the scrubbed data supplied by the doppelganger. This modification was only for test purposes, so that we could run the benchmark over our kernel. Even with this modification it was impossible to use SpecWeb99 on Apache with process swapping, since we could not completely eliminate the effect of data scrubbing; the swapped-in doppelgangers still had some scrubbed data in their buffers.

We configured SpecWeb99 to request static content of varying sizes. Figure 6 shows the server throughput as a function of the number of clients, and Figure 7 shows the response time. Our results show that the overhead of handling sensitive files was within 5%. The above graphs show that the saturation point for both configurations was in the range of 110–130 clients. These results further demonstrate that doppelgangers can provide privacy protection at negligible performance cost.

7 Related Work

Several recent system designs have observed the trouble that users and organizations have managing their sensitive data [3, 25, 29]. RIFLE [29] and InfoShield [25] both propose new hardware support for information-flow analysis and enforcement; SANE [3] enforces capabilities in-network. All of these approaches are orthogonal to TightLip. An interesting direction for our future work will be to design interfaces for exporting sensitivity between these layers and TightLip.

A simple way to prevent leaks of sensitive data is to revoke the network write permissions of any process that reads a sensitive file. The problem is that this policy can needlessly punish processes that use the network legitimately after reading sensitive data. For example, virus scanners often read sensitive files and later contact a server for new anti-virus definitions while Google Desktop and other file indexing tools may aggregate local and remote search results.

A number of systems perform information-flow analysis to transitively label memory objects by restricting or modifying application source code. Static solutions compute information-flow at compile time and force programmers to use new programming languages or annotation schemes [19, 26]. Dynamic solutions rely on programming tools or new operating system abstractions [10, 14, 32]. Unlike TightLip, both approaches require modifying or completely rewriting applications.

It is also possible to track sensitivity without access to source code by moving information flow functionality into hardware [6, 8, 28, 29]. The main drawback of this work is the lack of such support in commodity machines. An alternative to hardware-level tracking is software emulation through binary rewriting [5, 7, 20]. The main drawback of this approach is poor performance. Because these systems must interpose on each memory access, applications can run orders of magnitude more slowly. In comparison, TightLip's use of doppelgangers runs on today's commodity hardware and introduces modest overhead.

A recent taint checker built into the Xen hypervisor [13] can avoid emulation overhead as long as there are no tainted resident memory pages. The hypervisor tracks taint at a hardware byte granularity and can dynamically switch a virtual machine to emulation mode from virtualized mode once it requires tainted memory to execute. This allows untainted systems to run at normal virtual machine speeds.

While promising, tracking taint at a hardware byte granularity has its own drawbacks. In particular, it forces guest kernels to run in emulation mode whenever they handle tainted kernel memory. The system designers have modified a Linux guest OS to prevent taint from inadvertently infecting the kernel stack, but this does not address taint spread through system calls. For example, if email files were marked sensitive, the system would remain in emulation mode as long as a user's email remained in the kernel's buffer cache. This would impose a significant global performance penalty, harming tainted and untainted processes alike. Furthermore, the tainted data could remain in the buffer cache long after the tainted process that placed it there had exited.

TightLip's need to limit the sources of divergence after scrubbed data has been delivered to the doppelganger is similar to the state synchronization problems of primary/backup fault tolerance [1]. In the seminal primary/backup paper, Alsberg describes a distributed system in which multiple processes run in parallel and must be kept consistent. The *primary* process answers client requests, but any of the *backup* processes can be swapped in if the primary fails or to balance load across replicas. Later, Bressoud and Schneider applied this model to a hypervisor running multiple virtual machines [2]. The main difference between doppelgangers and primary/backup fault tolerance is that TightLip deliberately induces a different state and then tries to eliminate any future sources of divergence. In primary/backup fault tolerance, the goal is to eliminate *all* sources of divergence.

Doppelgangers also share some characteristics with speculative execution [4, 21]. Both involve "best-effort" processes that can be thrown away if they stray. The key difference is that speculative processes run while the original is blocked, while doppelgangers run in parallel with the original.

8 Conclusions

Access control configuration is tedious and error-prone. TightLip helps users define what data is sensitive and who is trusted to see it rather than forcing them to understand or predict how the interactions of their software packages can leak data. TightLip introduces new operating system objects called doppelganger processes to track sensitivity through a system. Doppelgangers are spawned from and run in parallel with an original process that has handled sensitive data. Careful monitoring of doppelganger inputs and outputs allows TightLip to alert users of potential privacy breaches.

Evaluation of the TightLip prototype shows that the overhead of doppelganger processes is modest. Data transfer micro-benchmarks show an order of magnitude better performance than similar taint-flow analysis techniques. SpecWeb99 results show that Apache running on TightLip exhibits a negligible 5% slowdown in request rate and response time compared to an unmodified server environment.

Acknowledgements

We would like to thank the anonymous reviewers and our shepherd, Alex Snoeren, for their valuable insight. We would also like to thank Jason Flinn, Sam King, Brian Noble, and Niraj Tolia for their early input on this work.

References

[1] P. A. Alsberg and J. D. Day. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the Second International Conference on Software Engineering (ICSE)*, October 1976.

[2] T. C. Bressoud and F. B. Schneider. Hypervisor-Based Fault-Tolerance. *ACM Transactions on Computer Systems (TOCS)*, February 1996.

[3] M. Casado, T. Garfinkel, A. Akella, M. J. Freedman, D. Boneh, N. McKeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proceedings of the 15th USENIX Security Symposium*, July 2006.

[4] F. Chang and G. A. Gibson. Automatic I/O Hint Generation Through Speculative Execution. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI)*, Feburary 1999.

[5] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. TaintTrace: Efficient Flow Tracing with Dynamic Binary Rewriting. In *Proceedings of the 11th IEEE International Symposium on Computers and Communications (ISCC)*, June 2006.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *Proceedings of the 13th USENIX Security Symposium*, August 2004.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[8] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2004.

[9] T. Dierks. The TLS protocol. Internet RFC 2246, January 1999.

[10] P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Maziéres, F. Kaashoek, and R. Morris. Labels and Event Processes in the Asbestos Operating System. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, October 2005.

[11] J. T. Giffin, S. Jha, and B. P. Miller. Efficient Context-sensitive Intrusion Detection. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2004.

[12] N. S. Good and A. Krekelberg. Usability and Privacy: a Study of Kazaa P2P File-sharing. In *Proceedings of the Conference On Human Factors in Computing Systems (HCI)*, April 2003.

[13] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical Taint-based Protection using Demand Emulation. In *Proceedings of the First EuroSys Conference*, April 2006.

[14] L. C. Lam and T. Chiueh. A General Dynamic Information Flow Tracking Framework for Security Applications. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, December 2006.

[15] J. Leyden. ChoicePoint Fined $15m Over Data Security Breach. The Register, January 27, 2006.

[16] J. Leyden. HK Police Complaints Data Leak Puts City on Edge. The Register, March 28, 2006.

[17] A. McCue. CIO Jury: IT Bosses Ban Google Desktop Over Security Fears. silicon.com, March 2, 2006.

[18] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Rreliability of UNIX Utilities. *Communications of the ACM*, 33(12), 1990.

[19] A. C. Myers. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[20] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, February 2005.

[21] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative Execution in a Distributed File System. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, October 2005.

[22] A. Press. Miami University Warns Students of Privacy Breach. Akron Beacon Journal, September 16, 2005.

[23] M. Rinard, C. Cadar, D. Dumitran, D. M. Roy, T. Leu, and W. S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[24] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *Selected Areas in Communications, IEEE Journal on*, 21(1), January 2003.

[25] W. Shi, J. B. Fryman, G. Gu, H. H. S. Lee, Y. Zhang, and J. Yang. InfoShield: A Security Architecture for Protecting Information Usage in Memory. In *Proceedings of the 12th International Symposium on High-Performance Computer Architecture (HPCA)*, February 2006.

[26] V. Simonet. Flow Caml in a Nutshell. In *Proceedings of the First APPSEM-II Workshop*, March 2003.

[27] B. Sprunt. Pentium 4 Performance Monitoring Features. *IEEE Micro*, July-August 2002.

[28] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure Program Execution via Dynamic Information Flow Tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, October 2004.

[29] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An Architectural Framework for User-Centric Information-Flow Security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (Micro)*, December 2004.

[30] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[31] A. Yumerefendi, B. Mickle, and L. P. Cox. TightLip: Keeping Applications from Spilling the Beans. Technical Report CS-2006-7, Computer Science Department, Duke University, April 2006.

[32] S. Zdancewic, L. Zheng, N. Nystrom, and A. C. Myers. Untrusted Hosts and Confidentiality: Secure Program Partitioning. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, Banff, Canada, October 2001.

Peering Through the Shroud:

The Effect of Edge Opacity on IP-Based Client Identification

Martin Casado and Michael J. Freedman
Stanford University
`http://illuminati.coralcdn.org/`

Abstract

Online services often use IP addresses as client identifiers when enforcing access-control decisions. The academic community has typically eschewed this approach, however, due to the effect that NATs, proxies, and dynamic addressing have on a server's ability to identify individual clients.

Yet, it is unclear to what extent these edge technologies actually impact the utility of using IP addresses as client identifiers. This paper provides some insights into this phenomenon. We do so by mapping out the size and extent of NATs and proxies, as well as characterizing the behavior of dynamic addressing.

Using novel measurement techniques based on active web content, we present results gathered from 7 million clients over seven months. We find that most NATs are small, consisting of only a few hosts, while proxies are much more likely to serve many geographically-distributed clients. Further, we find that a server can generally detect if a client is connecting through a NAT or proxy, or from a prefix using rapid DHCP reallocation. From our measurement experiences, we have developed and implemented a methodology by which a server can make a more informed decision on whether to rely on IP addresses for client identification or to use more heavyweight forms of client authentication.

1 Introduction

Faced with an ever increasing amount of unwanted traffic, Internet services must be able to differentiate clients in order to enforce access-control decisions. One traditional approach is to use the client's IP address as an identifier. For instance, if a client is exhibiting malicious behavior—such as attempting to post spam messages to online blogs or trolling ssh servers for weak passwords—a server may add the client's IP address as a firewall rule that will prohibit further connections from the offending address.

Unfortunately, middleboxes (both layer-3 network address translation (NAT) devices and layer-5 proxies) and dynamic IP renumbering due to DHCP challenge the utility of using IP addresses as client identifiers. Given the previous example, adding a firewall rule for an IP address due to abusive behavior (blacklisting) can deny system access to many users that may share that IP, either simultaneously via a middlebox or over time via DHCP. Similarly, when IP addresses are naively used to allow access to a resource (whitelisting), an open proxy from an authenticated domain, such as a university, can enable Internet-wide access to the protected resource. We refer to this phenomenon of edge technologies obscuring a client's identity at the network level as *edge opacity*.

Due to these apparent operational issues, many websites have moved from using IP addresses as identifiers to requiring some form of registration, authentication, and then application-level identity checks. For example, it is not uncommon for wikis or web forums to require their users to register, login, and then present HTTP cookies on each access. Yet, these extra steps can be a usability hurdle and generally require more server-side resources.

Because of these concerns, other websites continue to use IP whitelisting and blacklisting as the basis for managing client access. And yet, as the extent of edge opacity remains unquantified, the impact of using IP addresses as client identifiers is uncertain. Thus, server operators are making uninformed decisions regarding the best way to use IP addresses as client identities.

The goal of our work is to help a server determine if the IP address of an incoming client is a useful identifier for access-control decisions. We do so by exploring the extent to which edge opacity obscures a server's view of its clients. Specifically, we consider the following questions:

1. To what extent does edge opacity from NATs, proxies, and DHCP prevent accurate IP-based client identification?
2. Is it possible for a server to determine when IP-based filtering is insufficient and stronger methods of identification are required?

To answer the first question, we performed a large-scale study measuring 7 million unique web clients from 177 million HTTP requests over a seven-month period. We found that while the majority of clients are behind NATs (roughly 60%), NATs are small and localized—with most serving fewer than 7 hosts. Furthermore, clients using DHCP generally rotate their IP addresses on the order of several days, limiting the impact of dynamic addressing on short-lived access-control decisions. Web proxies,[1] on the other hand, while fewer in number (roughly 15%), often serve large and commonly geographically-diverse client populations.

To answer the second question, we introduce, analyze, and implement a set of techniques with which a web server can detect with high accuracy if a client is using a proxy or NAT, or whether it belongs to a prefix with rapid IP reallocation. Because of their potential to host large client populations, we mainly focus on proxy detection, proposing additional methods to determine a client's distance from its proxy and to disambiguate multiple clients behind a single proxy.

The main contributions of this paper are as follows.

- We introduce and demonstrate the utility of using active content as a measurement vehicle for understanding the Internet's edge (§3).
- We present the results of a large-scale measurement study which helps to quantify the extent and characteristics of NATs, proxies, and DHCP usage (§4). We believe these results are original and have independent academic interest.
- From our measurement experience, we derive and implement a methodology to aid Internet services in determining whether or not a client's IP address is a useful identifier (§5).

2 Background and Limitations

2.1 IP vs. application-level identification

There are a number of practical benefits to using IP addresses as the basis for enforcing access controls in today's Internet. IP-based filtering, ACLs, and rate-limits are all standard on firewalls and routers. Further, these IP-based enforcement mechanisms typically operate at carrier-grade line speeds. Filtering on IP addresses also allows an online service to "drop early," rather than waste CPU, memory, or bandwidth resources on traffic that will be eventually dropped via heavier-weight mechanisms.

Application-level identification methods (greylisting), on the other hand, can be used to differentiate clients when IP addresses are not sufficient. However, these methods often come at a cost. Many greylisting methods put a burden on users, for example, requiring them to answer CAPTCHAs [32] or to go through a separate sign-on step. Application-level identification is also more resource intensive for the server. Whereas IP blacklisting can drop packets before they ever reach the server, registration requires that the server handles a client's session prior to its decision whether to accept the client.

We note that if the server's greylisting scheme is effective, an attacker has no incentive to masquerade as a proxy. Most obviously, greylisting may require the attacker to expend resources—*e.g.*, requiring some cryptographic computations or additional time for "tarpitted" TCP connections to complete—better spent attacking other systems. Furthermore, if the time required to identify malicious behavior and blacklist the client is greater than that needed to greylist a proxy, an attacker has a disincentive to appear as a proxy: immediate greylisting would provide the attacker with fewer opportunities to attack the system. This argument does not necessarily hold if the attacker is attempting to launch a denial-of-service attack on the server; we consider this case in §2.3.

2.2 Other uses for edge opacity detection

With the exception of this subsection, this paper only discusses using IP addresses to enforcing access controls. In practice, IP addresses are used by servers in a variety of other settings, including geolocation and fraud detection, which we review here.

Websites use geolocation in both content personalization and access control contexts. For example, Major League Baseball uses Quova's GeoPoint server [20] to ensure that games are not webcast to subscribers subject to blackout restrictions. Yet end-users can often circumvent such restrictions by using proxies (indeed, we have personal experience with clients using CoralCDN for precisely this reason [3]); accurate proxy detection is needed to prevent such behavior. Furthermore, if a server can peer through a client's proxy to determine its true location (§5.1), advertisements can be more accurately targeted.

An important aspect of combating online fraud is to detect suspicious IP-based usage: *e.g.*, when a California-based bank customer accesses his online account from Romania, or when huge numbers of clicks for pay-per-click advertisements originate from a small IP range. Proxy detection can improve servers' decisions; for instance, by discovering when a fraudster is trying to proxy its banking session behind a U.S.-based IP address, or by recognizing the potential legitimacy of high rates of traffic originating from AOL's proxies. While efforts to correlate anomalous IP characteristics to fraud exist [10], their ability to detect and peer behind proxies is limited.

Our paper does not further address these two additional uses of IP-based identification. Indeed, evaluating the accuracy of a geolocation package or the security of a fraud-detection system is challenging and beyond the scope

[1] In this paper, we refer to address translation middleboxes that do not perform TCP termination as NATs. Layer-5 proxies, on the other hand, establish application sessions with both their clients and remote servers and therefore terminate their clients' TCP sessions.

Country	Unique hosts
United States	1,290,928
Japan	1,024,088
China	912,543
Germany	385,466
Great Britain	218,125
France	215,506
Canada	204,904

Table 1: Top 7 countries by unique hosts measured

of this paper. We describe these examples, however, to demonstrate why—if one were to deploy a geolocation or fraud-detection system (and many companies do exactly that [10, 12, 20, 26])—detecting proxies and other IP exceptions is important.

2.3 Limitations

In what follows, we present results from our efforts to characterize NAT, proxy, and DHCP usage and their effects on client identification. From March through September, 2006, our measurement servers handled over 177 million HTTP requests, originating from nearly 7 million unique hosts from 214 countries. Table 1 gives a sense of the dataset's international coverage, including large numbers of requests from the United States, Japan, China, and Europe. As we mention in 3.2, we collected measurements from web clients as they visited a wide variety of different sites serving largely different client demographics.

While these measurements are over a large and diverse population, we do not claim to provide a representative sampling of *all* Internet traffic.[2] Rather, our goal is to understand edge opacity from the viewpoint of a server operator: What is the extent to which its clients are behind middleboxes or experience rapid IP reallocation? Secondly, given this large body of measurement data, what techniques and heuristics can we develop to identify when a server operator should not use IP address and client configuration information for access control decisions?

We do not envision opacity detection to be useful to prevent denial-of-service attacks. An attacker with sufficient resources can resort to flooding with spoofed IP addresses, making any IP-based logic meaningless. Further, an attacker attempting an application-level DDoS could masquerade as a proxy and potentially afford herself more access to resources. However, we found that IP blacklisting will affect far fewer clients than we had originally anticipated (§4).

[2]For example, users of Internet cafes are probably more likely to use online email, chat, or game services, none of which we were able to measure.

3 Measurement Overview

3.1 Using active content for measurement

Our approach to studying the impact of NATs, proxies, and DHCP-based IP allocation relies on a variety of network- and application-level client characteristics. These include such properties as public and local IP addresses, round-trip-time measurements, system configuration parameters, and so forth. Unfortunately, much of this information cannot be captured through traditional measurement means alone, *i.e.*, active probing or passive traffic sniffing [7].

Instead, we leverage active web content as a means for gathering edge-network configuration state. We redirect oblivious web clients to a custom-built web server that serves web pages with active content (currently, javascript and/or a Java applet). The server then performs some limited passive analysis and cooperates with the executing code to measure and collect client configuration state, which is analyzed and added to our measurement database.

In contrast to active measurement, active content is executed at the application layer, generally *behind* NATs or proxies. Active content is in wide use by popular websites today; for example, users of Google's GMail application execute tens of thousands of lines of javascript. Thus, it is both almost universally supported and, at the same time, much less likely to generate abuse complaints normally associated with unsolicited scans of end-hosts [18]. (We have yet to receive a single abuse complaint.)

Furthermore, in contrast to expecting clients to download proprietary measurement code [13, 19], we target content providers. Web clients access our measurement platform either when requesting a web-beacon placed on popular websites (as of September 2006, 32 third-party sites have incorporated this web-beacon) or after being redirected from CoralCDN [9, 17]. While browser-based active content can collect strictly less information than client-executed proprietary code—insofar as it must operate within the confines of browsers' security policies—it is easier to gather large-scale measurements. For instance, the Dimes project [13] has collected measurements from roughly 11,000 agents from 5,000 users over 2 years of operation, almost three orders of magnitude less than our coverage.

3.2 Data collection methods

To cast a wide net, we collect information from two main sources. First, we built an online measurement community of content providers, allowing websites to track and compare their individual contributions. A wide variety of websites contributed to our measurements, including popular blogs, news sites, personal web-pages, education sites, hosting services, hobbyist sites, and public forums.

Second, we have instrumented CoralCDN [9, 17], a popular web content distribution network we run, to redirect a small percentage of its approximately 25 million daily requests through our measurement servers. Traffic from CoralCDN currently accounts for approximately two-thirds of our clients and its world-wide distribution.

Operators of servers can contribute to our measurement project in two ways. First, websites insert a transparent, embedded object into their web pages (the 1x1 pixel `iframe` "web-beacon" in Figure 1). When clients download these sites' pages, they subsequently request this web object from our measurement servers. Alternatively, a server integrates client-side measurements by redirecting link click-through traffic to our measurement servers, which causes clients to load our web-beacon before they are redirected back to the link's real destination. We used this latter approach for measuring clients of CoralCDN.

Whether through the web-beacon or a redirect, the client executes javascript that collects various configuration parameters, *e.g.*, browser version information, screen parameters, system languages, timezone settings, etc. Java-enabled clients may also execute a Java applet that creates a socket connection back to our measurement servers, from which the client grabs its local IP address and ephemeral port. Differences between the client's local IP address and its public IP address (as seen by our server) indicates the existence of an on-path middlebox.[3]

Of course, both the javascript and Java applet code are constrained by browser security policies, such as same-origin restrictions. Additionally, cognizant of privacy concerns, we have avoided collecting particularly invasive information (such as extracting browser cache history through CSS tricks).

Rather than explain the details of all our data-collection techniques upfront, we will describe these techniques incrementally as we consider various questions throughout the remainder of this paper.

3.3 Dataset coverage

This paper's dataset, described in Table 2, covers client data collected between March 3, 2006 through September 27, 2006. It includes nearly 7 million unique hosts, where a unique host is identified by a three-tuple of the client's public IP address, its local IP address if available, and its SYN fingerprint [27]. A smaller fraction of clients (1.1 million in all) executed our Java applet, due both to some third-party websites choosing a no-Java "diet" version of our web-beacon and the fact that only approximately 30% of our measured clients will actually execute Java.

[3]We can also use differences in the ephemeral ports to help detect transparent proxies and provide some insight into how the middleboxes are doing port re-mapping, although we do not include such analysis in this paper.

Unique targets	
Hosts measured	6,957,282
Public IPs	6,419,071
Hosts running Java	1,126,168
Hosts behind middleboxes	*73.8%*
Coverage	
IP Prefixes (per RouteViews)	85,048
AS Numbers (per RouteViews)	14,567
Locations (per Quova)	15,490
Countries (per Quova)	214

Table 2: Dataset statistics

In addition, we analyzed over 4 million requests from 540,535 clients that ran a heavier-weight version of the javascript code (collected between August 29 and September 27, 2006). This dataset, used in §4.3 and §5.1, included client configuration settings and used client-side cookies to link client sessions.

To assist with some analysis, we had access to Quova's geolocation database [20], the industry standard for IP geolocation. This database includes 31,080 distinct latitude/longitude locations for more than 6.85 million distinct IP prefixes. When we later refer to calculating the geographic distance between two IP addresses, we refer to first performing longest-prefix match to find the most appropriate IP prefix in this geolocation database, then computing the geographic distance between the latitude/longitude coordinates for these two prefixes.

As Table 2 shows, our dataset includes hosts from approximately two-thirds of all autonomous systems and more than one-third of all BGP-announced IP prefixes (per RouteViews [25]). It covers more than one-half of Quova's identified locations and 214 of 231 countries.

3.4 Summary of results

This section summarizes our two main results:

NATs and DHCP do not contribute largely to edge opacity from the viewpoint of a server. While the majority of clients are behind NATs (roughly 60% from our measurements), virtually all NATs are small. In fact, NAT sizes tend to follow an exponential, rather than a power law, distribution (§4.2). Second, IP reallocation due to DHCP is slow, generally on the order of days. For example, fewer than 1% of the clients we measured used more than one IP address to access our servers over the course of one month, and fewer than 0.07% of clients used more than three IP addresses (§4.3).

Proxies pose a greater problem for IP-based server decisions. While only 15% of clients we measured traversed proxies, these proxies were generally larger than NATs and often served a geographically-diverse client population (§4.4). This latter property is especially important if access control decisions are being made for regulatory compliance. Additionally, as a single client may

```
<iframe src='http://www.cdn.coralcdn.org/noredirect.html?teamid=fcc9c...0c7'
        scrolling=no frameborder=0 marginwidth=0
        marginheight=0 width=1 height=1></iframe>
```

Figure 1: Our measurement web-beacon

use many proxies in multiple locations, blacklisting proxy IP addresses in such cases is largely ineffective.

From these results, we conclude that a principal concern for server operators is the ability to detect and react to proxies. In §5, we introduce a set of classifiers with which a server, using only a single client request, can detect proxies with high accuracy.

4 The Extent of Edge Opacity

4.1 Establishing a comparison set

The analysis in the next two sections requires the ability to detect the existence of a middlebox, as well as to determine whether the middlebox is a NAT or proxy.

To do so, we first build a set of all measured hosts that have clear identifiers of having come through a NAT or a proxy. We characterized a client as having traversed a middlebox if its public IP address differed from the local IP addresses, as returned by our Java applet. (We note that those clients whose local and public IP address are the same were classified as *non-middleboxes*, a dataset used later in §5.1).[4]

We also built an independent comparison set consisting solely of proxies and another consisting solely of NATs. An IP address was considered a proxy if its request contained a standard proxy header, *e.g.*, `Via`. We classified a middlebox as a NAT iff (1) the SYN fingerprint was not a known proxy type, *e.g.*, Cisco or NetApp, (2) the request did not contain proxy headers, (3) the domain name of the middlebox was not suggestive of a proxy, *e.g.*, contained *proxy*, *prx*, *cache*, or *dmz*, and finally, (4) the ratio of distinct `User-Agent` strings to distinct SYN fingerprints was at most three (we explain the rational for this metric in §5.2.2). Under these criteria, we classified 128,066 public IPs as proxies and 444,544 as NATs.

We will also use these sets of known proxies and non-middleboxes in Section 5, in order to evaluate the accuracy of a number of lighter-weight proxy detection heuristics that do not rely on Java.

4.2 NAT characteristics

We begin by showing that the majority of NAT'd networks only consist of a few hosts. To establish this result, we leverage the fact that most devices assign addresses from their DHCP address pool in linear order, starting from some (usually well-known) default value. For example, the DHCP pool of Linksys routers starts at 192.168.1.100 [23]. Then, given the local IP addresses of our clients, we analyze the frequency histogram of the private address space usage between NATs.

Given the frequency histogram of the private address range, we have found that one can easily detect the default starting addresses used by DHCP servers. For example, Figure 2 shows a histogram of 192.168.0/24: There are two clear spikes starting at 192.168.0.2 and 192.168.0.100. Thus, given a common starting value (the beginning of a spike) and given that IP addresses are largely assigned in linear order, we can estimate the distribution of NAT'd network sizes that share the same starting IP address.

More specifically, a client's rank in a group of size n can be approximated by its IP address subtracted by the default starting address. We term this value a host's *IP rank*. In the limit, the average IP rank of measured hosts behind a NAT will converge to $\frac{n}{2}$. Given a sufficient sampling of hosts from a single public IP address, this could be used to approximate the NAT's size. Of course, this technique is useful only for a *particular* NAT and requires measurements from multiple hosts behind that NAT. Our dataset contained requests from multiple hosts in fewer than 10% of the NAT'd public IPs.

However, one can still reach useful conclusions given aggregate data across many NAT'd IPs. Recall that Figure 2 plots an aggregate histogram of the frequency with which each private IP address is used by unique hosts. (again, unique hosts are identified by a unique *(pub-ip, local-ip, SYN-FP)* tuple.) We can use this aggregate information to determine the relative sizes of individual NAT'd networks. To do so, we take the discrete derivative of the histogram, which approximates the ratios of relative NAT sizes. As a simple example, a derivative value of ten for IP rank 1 and derivative value of two for IP rank 2 implies that NAT'd networks with one unique host are five times more likely than networks with two hosts.

Figure 4 shows the discrete derivative calculated over the smoothed histogram[5] for IP address range 192.168.1.100 through 192.168.1.199, inclusive. We use this range for demonstrative purposes, as it contains a single clear spike (see Figure 2): this implies that most hosts within this range use the same starting IP address within their DHCP pool, *e.g.*, due to Linksys routers' default con-

[4]During our analysis, we uncovered many *transparent* web proxies which do not perform address translation. However, since these do not render the client's IP address opaque, we treated them in the same manner as client requests which did not traverse a middlebox.

[5]Using a degree-five Bernstein basis polynomial for smoothing [24].

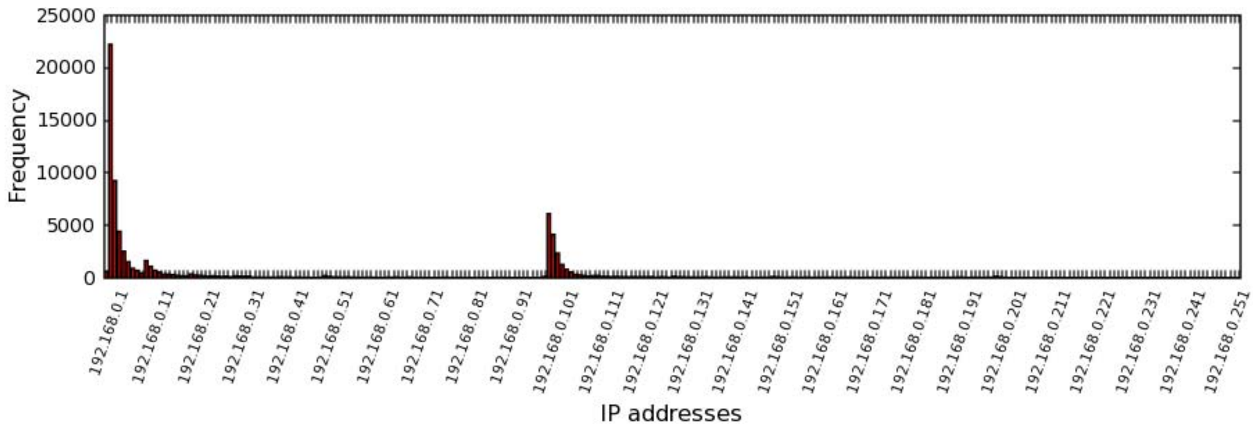


Figure 2: Histogram of the IP address usage within the 192.168.0/24 prefix of all clients which ran the Java Applet

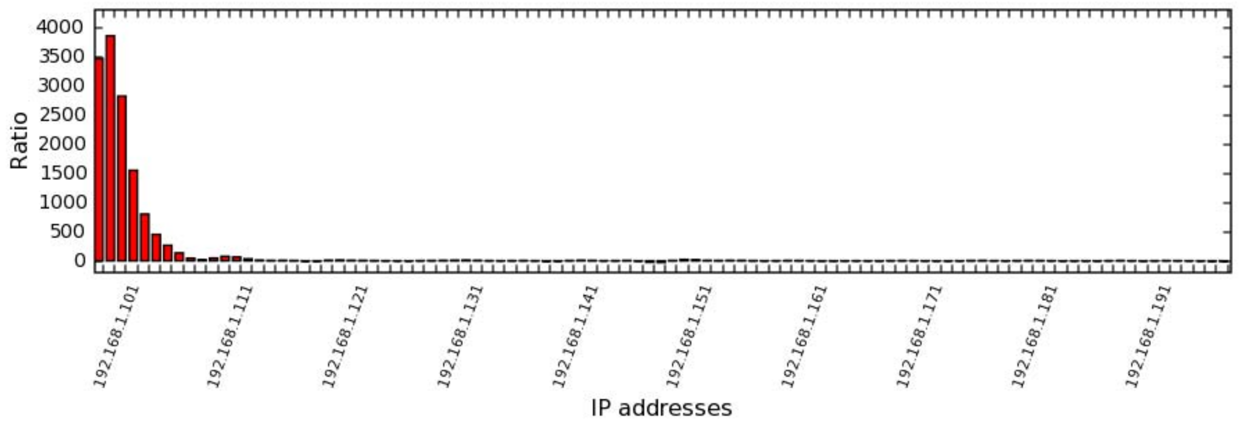


Figure 3: Discrete derivative of the smoothed frequency histogram for 192.168.1.1xx

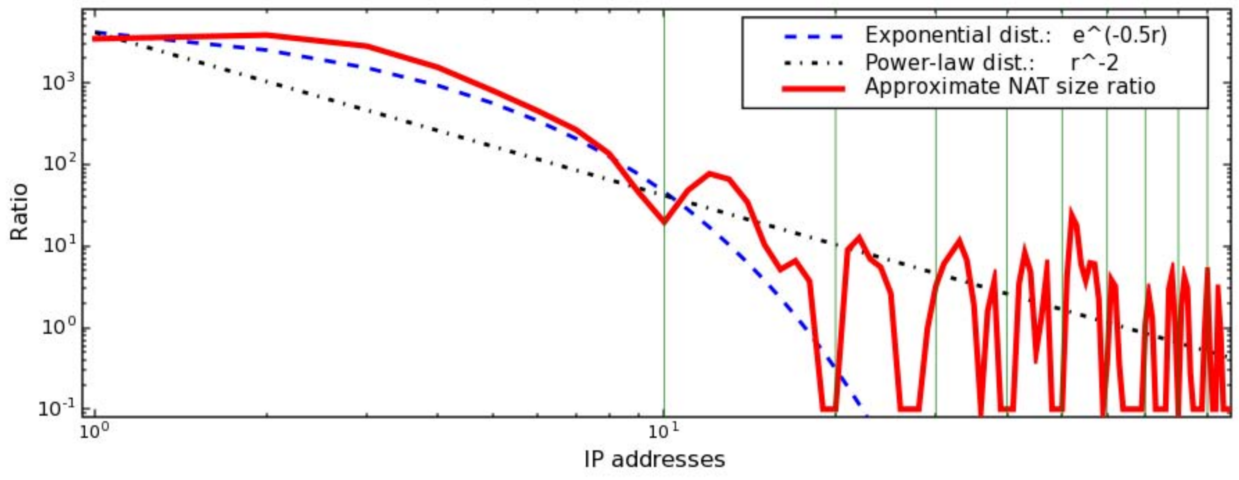


Figure 4: Discrete derivative of the smoothed frequency histogram for 192.168.1.1xx on a log-log scale, with common exponential and power-law curves plotted as comparison. Vertical lines are drawn every 10 IP addresses.

figurations [23]. With the derivative peaking at IP rank 2, we conclude that for this IP range, the most common NAT'd network has two hosts. In fact, the vast majority of networks in this range have fewer than 7 hosts, and there are almost 200 times fewer networks of 10 hosts than those of 2 hosts (IP rank 10 has value 19.6).

Figure 4 plots the same discrete derivative on a log-log scale, given as the solid line. We see that the curve is somewhat periodic, with new (yet smaller) peaks roughly correlating to intervals of ten IP addresses. While points at increasing IP rank may be caused by the existence of (outlying) large networks, this periodicity likely indicates less-common defaults used by some DHCP pools for numbering a network.

Figure 4 also includes an exponential and a power-law distribution curve that attempt to fit the data. The plot suggests that NAT'd networks follow an exponential distribution: The first curve of IP address usage starting at 192.168.1.100—before the periodic jump at 192.168.1.110—closely follows $e^{\frac{-r}{2}}$.

We applied this NAT-size analysis to all other populated regions of the private address space that we measured, including 10/8, 192.168/16, 172.16/12, and 169.254/16. Overall, the characteristics of the analyzed IP ranges appeared similar to the results presented above. One exception is that 10/8 had (only slightly) largely NAT sizes, yet we also found an order of magnitude fewer hosts using 10/8 networks compared to 192.168/16 networks. We do not include the graphs of these results due to length limitations.

Analyzing the data less in aggregate, of the 444,544 NAT'd networks we measured, only 112 networks have more than 10 hosts (or $\leq 0.03\%$). The largest NAT'd networks we discovered were heavily biased based on geographic location: Of the largest ten networks, eight were residential ISPs in Japan (including all top-five), with one additional ISP in China and one for Google. As we show in Section 5, it is possible for a server to detect and track exceptionally large NATs.

4.3 DHCP usage characteristics

In this section, we analyze dynamic IP renumbering. To do this, we track clients using HTTP cookies. All results presented in this section were derived from a one month subset of our data in which we added javascript to perform cookie management. The dataset covers 537,790 individual clients (where clients are identified by unique cookies) across 4 million web sessions.

Figure 5 shows the extent to which clients use multiple IP addresses over time. Each histogram bar for time epoch i includes those clients whose first and last accesses to our servers are fully contained within the time period $[i, i+1)$. Each bar additionally shows the breakdown of the number of IP addresses used by its respective client

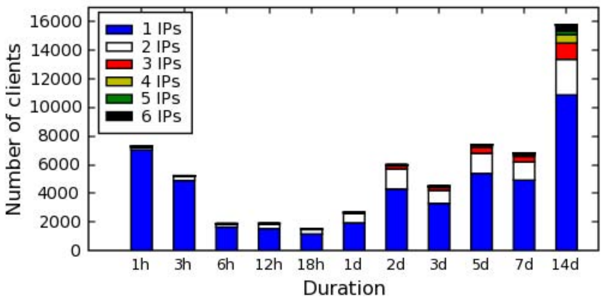


Figure 5: Prevalence of multiple IP usage by clients over time

Clients affected	All public IPs	Proxy IPs only
2	9.7%	53.4%
3	5.1%	40.2%
5	3.1%	28.3%
10	1.4%	17.3%

Table 3: Probability of collateral damage when blacklisting public IP addresses.

over its lifetime. We excluded only those clients who use more than one public IP address within 10 minutes—hand verification shows such clients to be using some form of load-balancing or anonymizing proxies.

We find that fewer than 2% of clients use more than 2 IP addresses between 3-7 days, with only 8% of clients using more than 3 IPs in 2-4 weeks. In fact, 72% of nodes used a single IP address (and 91% used 2 IPs) to access our servers for up to 2 weeks! Thus, we conclude that DHCP allocation results in little IP renumbering of clients from a server's perspective.

If clients switch IP addresses relatively rarely, can a server simply blacklist an IP address and not worry about the potential collateral damage to other hosts that may be effected by this filtering rule? In other words, how often would other clients re-use the same IP address already blacklisted?

Table 3 shows the extent of this collateral damage over the course of the dataset's month duration. We find that over 90% of the time, blacklisting any IP address would not have effected any other clients. Only in 1.4% of the cases would it effect more than 10 clients. In fact, the damage may even be less, as this analysis uses cookies as a form of identity. Clients may delete their cookies and thus inflate the measured damage. Indeed, other metrics of client identity—such as generating a client fingerprint based on a large set of browser configuration settings, many of which have significant entropy—appear to yield better preliminary results, although we do not include such analysis due to space limitations.

Finally, Table 3 demonstrates the importance of handling proxies differently from NAT'd and non-NAT'd IP addresses. We see that blacklisting a proxy would have some collateral damage 53% of the time and impact more than 10 clients in over 17% of cases.

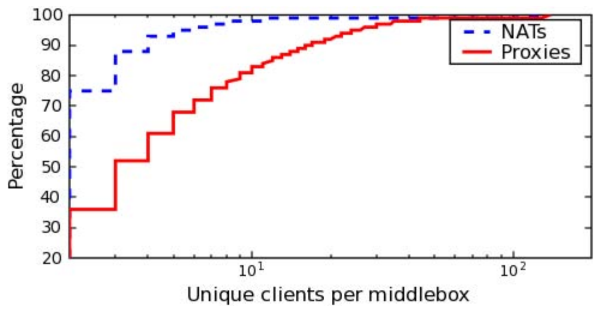


Figure 6: CDFs of the number of unique hosts behind each NAT and proxy devices

4.4 Proxy characteristics

This section shows that proxies both tend to be larger than NATs and often serve clients with high geographic diversity. These results are unsurprising: while virtually all home routers support NAT, proxies are often deployed in support of larger organizations. Further, unlike most NATs (VPNs excepted), proxies do not need to be on route and thus can serve clients from anywhere on the Internet.

Per §4.1, we created a set of *known proxies* by including the public IP addresses of those clients whose requests contained popular proxy headers. All IPs characterized as proxies also must have had a least one client that ran Java to reveal the existence of address translation.

Proxy sizes. Many of the larger proxies' clients contain publicly-routable local addresses. Unfortunately, this makes it difficult to estimate proxy sizes using the same DHCP allocation insight from §4.2. Instead, we plot the number of unique clients which accessed our system through proxies versus those using NATs (Figure 6). This figure only includes middleboxes in which we identified two or more distinct clients.

The graph shows that, as viewed from our servers, proxies serve a larger client population than do NATs. For example, only 10% of NATs were traversed by three or more hosts, opposed to 50% of proxies. While these results may not be representative for all servers, they do align with our intuition that proxies are likely to represent larger client groups.

Client-proxy locality. We now explore the proximity between proxies and the clients they serve. This subsection's results are limited to those pairs for which the proxy's public IP address differs from the publicly-routable local IP address of the client. Our dataset includes 26,427 such pairs of clients and web proxies. All geographic location information is obtained from Quova[20].

Because proxies are often off-route, they have two characteristics not found in NATs. First, clients from globally-reachable proxies can originate from anywhere. Figure 7 demonstrates this by mapping out all clients accessing our measurement servers through a Google web proxy located

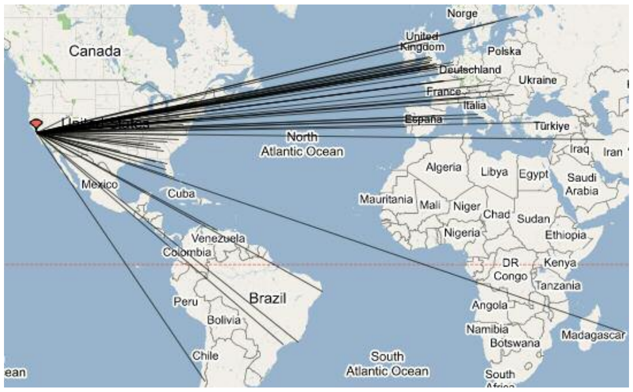


Figure 7: Location of clients accessing web via Google proxy (identified by marker)

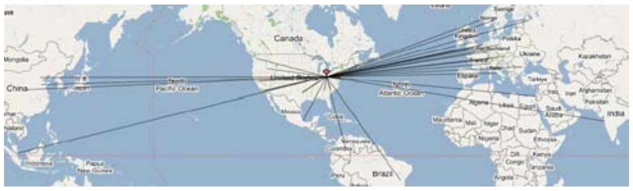


Figure 8: Location of anonymizing egress points used by clients within a single class-C network (identified by marker)

in Mountain View, CA. While a server operator may be comfortable whitelisting an IP shared by a large, albeit localized, NAT, she may wish to restrict unfettered access to such a proxy.

Secondly, while NATs virtually always map a small set of globally-reachable IP addresses to a larger number of machines, proxies are sometimes used to give a client access to multiple public IP addresses. For example, Figure 8 plots the actual location of colocated clients connecting to our measurement site through numerous proxies around the world. We believe these proxies to be part of an anonymization network. In such cases, where a single machine commands multiple geographically- and topologically-diverse IP addresses, attempting to curtail a particular client's access through IP blacklisting is near impossible.

From a more quantitative perspective, Figure 9 shows a CDF of the distance between clients and their web proxies. 51.4% of clients have the same location as their proxies. However, fully 10% of client-proxy pairs are located more than 3,677 kilometers apart, while 1% are located more than 10,209 kilometers apart. Table 4 summarizes locality properties with respect to countries and regions for our dataset. We also analyzed clients' logical proximity to the proxies they traverse. We found that only 29.2% of clients shared the same AS number as their proxy and only 6.7% shared the same IP prefix (per RouteViews BGP data [25]).

Finally, for completeness, we analyzed the routing inefficiency caused by clients' detour routes to their web

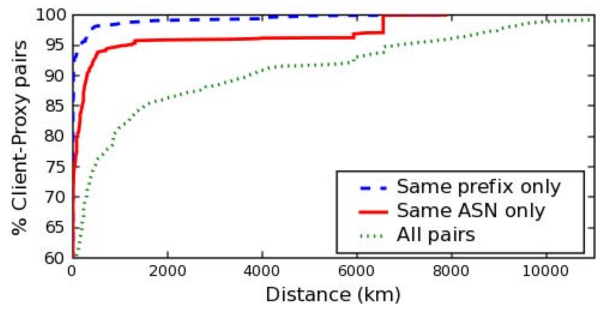


Figure 9: Proximity of web proxies to their clients

Country	City	Geolocation	SLD
85.1%	51.4%	51.4%	56.1%

Table 4: Percentage of client-proxy pairs located within the same country, city, or distinct location, or belonging to the same second-level domain (*e.g.*, *stanford.edu*).

proxies (Table 5). All destinations considered in this dataset correspond to our measurement servers, all located in Stanford, CA.

While these results do not directly impact edge opacity, it does provide some insight into the degree with which clients go out of their way to access content. The choice of longer detour routes may be forced on customers by ISPs that choose to deploy fewer sets of proxies, *e.g.*, for easier management. For example, much of AOL's European web traffic exits its network at proxies located in Frankfurt, Germany [1]. On the other hand, longer detour routes may be chosen by clients purposefully, either for anonymization or to skirt regulatory-compliance restrictions. In fact, we were able to detect numerous instances of clients using anonymizers (similar to that in Figure 8), making them normally appear as multiple clients distributed world-wide.

5 Middlebox Detection

We now present and analyze a set of techniques for detecting the usage of NATs and proxies. We first describe a toolkit of classifiers which can be used to detect proxies in real-time. We then describe how a server can use client history to determine the IP addresses of large NATs, as well as prefixes with high rates of IP renumbering among its clients. Finally, we describe our implementation of a custom web-server that implements such techniques.

5.1 Real-time proxy detection

The previous section's results suggest that proxies have a higher potential of negatively impacting server decisions. Not only are their client populations geographically distributed, but, unlike with NATs, a single client can hide behind multiple proxies. Fortunately, we find that it is

	50%	75%	90%	99%
Direct distance (km)	3890	8643	10611	16908
Detour distance (km)	4135	9156	13038	18587
Stretch	1.06	1.06	1.23	1.10

Table 5: Comparing direct vs. detour distances of web proxies

possible for a server to detect proxies in real time, *i.e.*, immediately following a single client request.

To do so, we introduce and analyze the following set of proxy indicators: (1) checking HTTP headers, (2) SYN fingerprinting clients, (3) geolocating clients, (4) comparing client timezone against geolocation information, (5) comparing client language against geolocation, and (6) analyzing application versus network round-trip-times. In what follows, we analyze the efficacy of each technique as well as their ability to detect proxies in the aggregate. Our analysis uses client configuration state returned by active content, passive server-side analysis, limited active probing, and IP geolocation (specifically, Quova's package [20]).

It is also possible to detect proxies using more intrusive techniques. One common method is to attempt to connect back to an incoming client's IP address at known proxy ports. Obviously, this technique only works for public proxies. Another approach is to use a client-side Java applet that issues a GET request for a popular object from a different domain than the server, *e.g.*, the banner image for `google.com`. A successful request would imply a cache hit on an on-path proxy. In this section, however, we limit our evaluation to the former six approaches, as they are both light-weight and have low visibility.

To evaluate the utility of each heuristic, we test them against the 22,546 known proxies per §4.1, as well as those clients classified as non-NAT'd hosts (non-proxies). A false positive (FP) occurs when a technique classifies a non-proxy as a proxy, while a false negative (FN) corresponds to a proxy being classified as a non-proxy. We summarize the results of each classifier for this *headers* dataset in Table 6.

Table 6 also evaluates our classifiers against an alternative *dns* dataset of proxies: The set of middleboxes having a domain name suggestive of a web proxy (*e.g.*, containing *cache*). While this domain-name test does not provide the certainty offered by proxy headers, it provides another useful testing set.

Overall, we find that our combined classifiers can detect a proxy between 75% and 97% of the time, with a false positive rate between 1.9% and 2.5%. These numbers merely serve to qualify the proposed classifier's effectiveness, as we combined them in a rather straight-forward manner. Certainly, a more intelligent weighting of individual classifiers' results (perhaps using machine learning techniques) should be able to improve the combined classifier's accuracy.

Classifier	ID'd proxy		False Negatives		ID'd non-proxy		False Positives	
dataset	headers	DNS	headers	DNS	headers	DNS	headers	DNS
Headers	n/a	94.9	n/a	5.1	n/a	100.0	n/a	0.0
SYN-FP	67.3	65.3	5.8	2.4	86.8	86.3	1.9	1.9
Geolocate	18.0	24.4	n/a	n/a	n/a	n/a	n/a	n/a
Timezone	6.1	5.0	n/a	n/a	n/a	n/a	2.0	2.0
Language	5.7	4.8	n/a	n/a	n/a	n/a	0.5	0.6
RTT	52.9	?	n/a	n/a	n/a	n/a	9.1	?
Combined	75.7	97.6	5.0	2.3	85.0	97.5	2.5	1.9

Table 6: Effectiveness of real-time classifiers for known proxies (in percentages) for both the header- and dns-based proxy datasets. *n/a* corresponds to a test being non-applicable; ? denotes insufficient data.

Proxy headers. Web proxies are required by the HTTP/1.1 specification (RFC 2616 [14]) to add their information to a `Via` header for both HTTP requests and responses that they proxy. Unfortunately, not all proxies do such, making proxy detection a non-trivial task.

Limiting our consideration to the *known-proxy* dataset—which by construction must include at least one proxy header—13.5% lack the required `Via` header. 69.6% of these proxies include an `X-Forwarded-For` header (a non-standard header introduced by Squid [30]) and 9.0% include `X-Bluecoat-Via` [6]. Although we tested for five other anecdotally-used headers (*e.g.*, `Client-IP`), we did not found any evidence of other such proxy headers in use.

On the other hand, if we consider the set of proxies in the *dns* dataset, we find that 94.9% include at least one proxy header. Of course, given their indicative hostnames, these proxies are not attempting to hide their identity, unlike many anonymizing proxies.

Client SYN fingerprinting. Our modified web server captures the SYN packet of all incoming requests and uses it to generate the client's SYN fingerprint [27]. SYN fingerprints (SYN-FP) provide an estimate of the sender's operating system. In the case of web proxies, which terminate their clients' TCP sessions, the SYN fingerprint corresponds to the proxy's TCP stack, not that of the clients. SYN fingerprints can be immediately used to uncover a variety of operating systems not commonly belonging to end-hosts (*e.g.*, Cisco, NetApp, and Tru64).

We can increase the coverage of this classifier, however, when combining SYN-FP information with operating system information returned in a client's HTTP `User-Agent` string. Specifically, we flag a host as a proxy if its SYN-FP host-type differs sufficiently from its `User-Agent` host-type, and mark it as a non-proxy if they are the same (although this latter step can certainly lead to false negatives).

Analyzed against our known proxies and non-proxies, this classifier successfully identified 67.3% proxies and 86.4% non-proxies, with a 2.3% false positive (FP) and 5.8% false negative (FN) rate. The classifier failed whenever the SYN-fingerprint did not match well-known operating-system characteristics. The higher FN rate is largely due to vagaries in the Windows TCP stack, yielding sometimes-inaccurate host-type estimations (*e.g.*, NT 5.0 vs. NT 5.1) from SYN fingerprinting.

Geolocation. Given a client's local IP address (from our Java applet) and its public IP address, we directly compare the geolocation information known about these two IP addresses. This method is limited to clients whose local IPs are globally reachable. But as Table 4 shows, these locations often differ. When classifying an IP address as a proxy if its public and local IP addresses differ in location, we are able to successfully identify 18.0% of our known proxies. We make no conclusion about those clients sharing the same location as their public IP address—while proxies may serve geographically-diverse clients, they can certainly serve local ones as well—hence the lower identification percentage. Additionally, given that our list of known non-proxies required matching local and public IP addresses, we cannot perform such analysis on non-proxies to generate an FP rate.

Client timezone. We now consider how client geolocation hints may differ from the corresponding information supplied by the server's geolocation package. This approach is unnecessary if a client can be geolocated directly (as described immediately above), but it provides a lighter-weight approach and may be applicable when a client's local IP address is in a private prefix.

Specifically, a web client exposes its timezone information in its `Date` header (as well as directly via javascript). Our timezone classifier compares client timezone information to that of its geolocated public IP address, over all client requests arising from that IP address. Unfortunately, this approach does not appear very useful, identifying only 6.1% of known proxies while yielding a 2.0% FP rate, given the low apparent rate of requests across timezones as compared to misconfigured or mobile hosts. Note that direct IP geolocation (as above) using a commercial database provides much higher granularity and thus higher accuracy.

Client language. Similar to timezone information, we see how client language information may serve as a geolo-

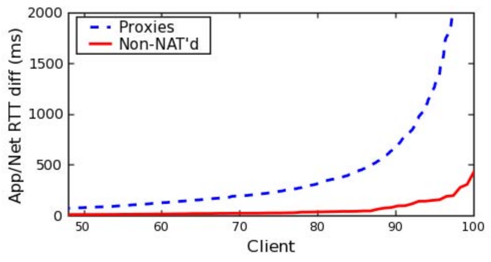


Figure 10: Differences in application- versus network-level RTT measurements between proxies and non-proxies

cation hint. Client language is collected via javascript, although it is also available per the `Accept-Languages` HTTP header. After building a database that maps languages to countries, we asked whether a client's IP address is located within a country using that language. If some sizable fraction of an IP address's clients cannot be matched via language, the IP address is marked as a proxy.

This language heuristic identified 5.7% of our known proxies, with a 0.5% FP rate. The coverage of this classifier is significantly limited by the universality of some languages—*e.g.*, browsers and hosts are configured with language settings of `en-us` world-wide, which we subsequently ignored—which led to our inability to make meaningful decisions on a majority of clients.

RTT analysis. Finally, we use differences in application-level and network-level round-trip-times (RTTs) to detect proxies. Specifically, we compare the difference between TCP RTT measurements on the server-side to the minimum request time over multiple HTTP GET requests issued by the client. We use javascript to perform the application-level RTT measurements (by timing asynchronous `XMLHttpRequests`).

A large difference between application- and network-level RTTs may suggest a client using a distant proxy. Figure 10 compares this difference as measured against our known proxies and non-proxies. While non-proxies can show a significant RTT gap—likely due to application-layer scheduling delays yielding inaccurate RTT measurements, even across multiple HTTP requests—this RTT gap is much more pronounced among clients of proxies.

This technique was able to identify 52.9% of our known proxies (using an RTT difference threshold of 50ms), although it had a much higher FP rate at 9.1% due to non-proxies experiencing significant application RTT delays. We draw no conclusion if a client has a similar network and application-level RTT.

Additionally, unlike some previous techniques which used client-supplied configuration state, this approach is less susceptible to malicious clients: a client that does not have control over the proxy it traverses can only make itself appear further from its proxy—and hence more likely be flagged as indeed behind a proxy—not closer.

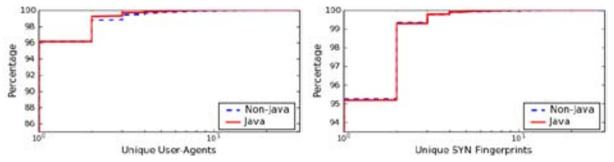


Figure 11: CDFs of unique User-Agent strings and SYN-FPs per public IP for hosts that did and did not run the Java applet

5.2 History-based middlebox detection

This section presents a methodology by which servers can identify large NATs and proxies, as well as distinguish between the two, by using the history of requests seen from a particular IP address or prefix. For this analysis, the server must record the public IP, SYN-FP, and `User-Agent` for each request.

We first show that non-NAT'd hosts show little variability in User-Agent strings and SYN fingerprints, as compared to clients behind middleboxes. Coupled with cookie usage, this heuristic provides a strong first-order differentiator of NAT'd and non-NAT'd hosts. Second, given that non-NAT'd hosts show little variability in these parameters, one can often differentiate between individual clients behind a middlebox using User-Agents alone, as the strings have a large amount of entropy. Finally, we determine the accuracy of detecting and differentiating between NAT'd and proxied networks by analyzing the distribution of User-Agents and SYN-FPs.

As before, we limit our study to those clients that ran the Java applet. However, as Figure 11 shows, the User-Agent and SYN-FP uniqueness characteristics of the entire dataset closely matches those of the subset of clients that ran the applet. Thus, we believe that our results generalize to the larger dataset.

5.2.1 Identifying NATs and proxies

How do the characteristics of non-NAT'd hosts differ from those of NAT'd hosts? One obvious approach to detect the presence of multiple clients behind a middlebox is to monitor a public IP address for any changes in its local hosts' configurations (*e.g.*, suggesting different operating systems or application versions). This approach is commonly discounted, however, due to dynamic IP addressing, routine changes to machine configurations, operating system or application updates, and even physical hosts that support multiple operating systems.

However, we have found that hosts identified as non-NATs—*i.e.*, hosts having the same local and public IP address—have a relatively low variability in User-Agents and SYN-FPs over time. Figure 12 shows the number of unique User-Agents for non-NAT'd hosts compared to those behind middleboxes. For the hosts behind middleboxes, we only include public IP addresses from which we identified two or more local IP addresses.

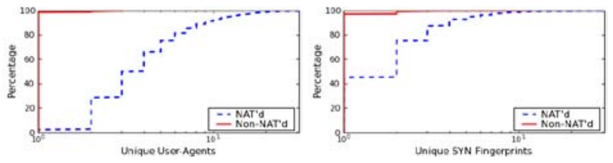


Figure 12: CDFs of unique User-Agents and SYN-FPs per public IP addresses for non-NAT'd hosts and NAT'd hosts

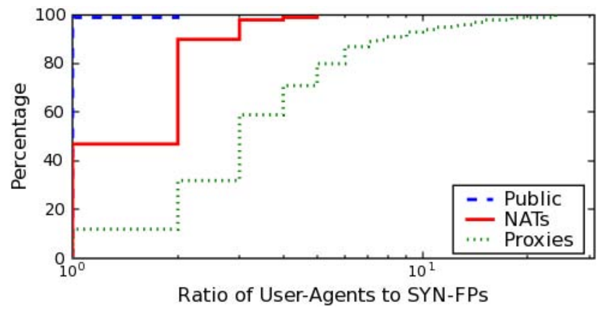


Figure 13: User-Agent to SYN-FP ratio for public (non-NAT'd), NAT'd, and proxied hosts

Fewer than 1% of the non-NAT'd hosts have multiple User-Agents, while the most any single host presented was three. Additionally, only 2.5% of such hosts had more than one SYN-FP, with a maximum of 11. While this does not show that non-NAT'd hosts are not being periodically renumbered via DHCP, it does corroborate our results in §4.3 that these rotations are sufficiently slow from a server operator's vantage. On the other hand, requests from middleboxes serving at least two clients had a much higher number of unique User-Agents and SYN-FPs.

This lack of different User-Agents from non-NAT'd addresses is not due to their paucity, however, as we found a total of 164,122 unique User-Agents among all clients. In fact, the frequency plot of these User-Agent strings follows a clear power-law distribution (we omit due to space limitations). This distribution suggests that there exists sufficient entropy to often differentiate between individual hosts' User-Agents.

5.2.2 Differentiating NATs and proxies

Given that the variability in User-Agents is a good first-order detector of middleboxes, we now turn to determining whether a middlebox is a NAT or proxy. Certainly, all the real-time methods introduced in §5.1 can be used to detect proxies. However, we also find that comparing the ratio of User-Agents to SYN-FPs for a particular IP address over multiple requests is useful for distinguishing between non-NAT'd, NAT'd, and proxied hosts.

Figure 13 plots the ratio of User-Agents to SYN-FPs for these three types of hosts (non-NAT'd, NAT'd and proxies). As before, we only include middleboxes from which we observed at least two clients. One finds a clear distinction between these three classes. For non-NAT'd hosts, the ratio is one in 99.99% of the cases — that is, in almost all cases, there is exactly one user-agent string per unique SYN-FP per IP. If the ratio is ≥ 3—that is, there are at least three distinct user-agent strings per SYN-FP—the middlebox is a proxy with high probability.[6]

5.3 Implementation

We have implemented the techniques discussed in this section into a custom web server which provides real-time proxy and NAT detection to aid in access-control decisions or other IP analytics applications. Our implementation consists of approximately 5,000 lines of C++, as well as the accompanying javascript and Java applets.

Figure 14 shows how our server, hereafter called the *illuminati-server*, integrates into an existing website. The website includes both a standard Web server (serving dynamic pages), an IP analytics engine, and the illuminati-server. In fact, we are currently integrating and deploying the illuminati-server alongside Quova's GeoDirectory Server [20] (the analytics engine in Figure 14) for commercial real-time proxy detection and geolocation.

A website integrates this solution as follows. First, the webserver dispatches client requests to the illuminati-server by adding an embedded `iframe` (similar to Figure 1) or `script` object to any pages for which it wishes to gather IP-based information (Step 1). This embedded object returned by the web server is tagged with a unique session id with which to later identify a particular client (as multiple clients may share the same public IP address). The client then fetches the embedded object from the illuminati-server, which subsequently serves an appropriate javascript or Java applet to the client (Step 2). This object is loaded in the background and therefore does not contribute to client-perceived latency on modern browsers.[7] After executing, this active content sends its client-generated information back to the illuminati-server, along with the server-specified session identifier (Step 3). Next, the illuminati-server pushes this information to the IP analytics engine (Step 4), which both stores them for historical analysis and, combining the data with IP geolocation information, analyzes whether the client is behind a NAT or proxy and, if the latter, where the client might be actually located. Depending on the results of this analysis, the IP analytics engine might call for other code to be executed by the client (*i.e.*, repeat Steps 2-4). Finally, the results of this analysis can be queried by the decision logic of the website, keyed by session identifier, at any point after it is collected (Step 5).

In this particular configuration, the time between a client's initial request and the time that a detection result

[6]We hand-inspected all middleboxes that the above criteria identified as NATs yet displayed a high User-Agent-to-SYN-FP ratio. From this, we identified 30 more large proxies missed using our initial criteria.

[7]Unless the Java virtual machine must be booted, which can cause some noticeable delay.

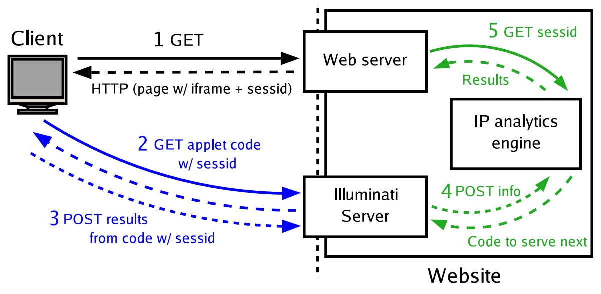


Figure 14: Integrating the illuminati server

may be available is at least 2 RTTs: half an RTT to return the initial page, 1 RTT to get the embedded code, and half an RTT to post the results. Of course, if application-level RTT tests are also used, each measurement adds one additional RTT. Serving additional applets increases this delay as well. Still, we expect that proxy detection would primarily be used for pages without strict latency requirements, such as to accompany user authentication.

Notice that this implementation provides for a separation of function between a simple stand-alone illuminati server and the more complex IP analytics engine. The illuminati-server must be deployed outside of any local reverse proxy, so that it receives direct TCP connections from a client (in order to get the client's public IP address, SYN headers for fingerprinting, and network RTT times). The illuminati-server and engine can also be hosted at some site external to the webserver, in which case the webserver would remotely query the engine over a secure channel. In both cases, the service can be integrated easily into existing websites.

6 Related Work

Architectural proposals. Many research proposals have offered new protocols or architectural changes which support globally identifiable end-hosts. These include HIP [22], DOA [33], LNA [2], UIP [15], IPNL [16]. TRIAD [8] and i3 [31]. These proposals generally focus on solving multiple architectural problems including edge opacity. One of the primary motivations is to decouple end-host identifiers from topology, allowing end-hosts to retain the same identity during mobility and multi-homing. However, globally unique end-host identifiers could also provide a basis for access-control decisions and thus ameliorate the ills caused by NATs.

Unfortunately, even if these architectures were to be adopted, proxies may continue to mask client identities. Proxies generally operate at the session layer, yet most of these proposals suggest adding a shim header between the IP and transport layer. Our findings suggest proxies are widely used for load balancing, caching, anonymizing, and skirting access controls. It is unclear how mechanisms operating below the transport layer address the issue of proxies.

IPv6 [11] confronts the problem of end-to-end client reachability by using 128-bit addresses. IPv6 also mandates support of IPSec, which, with the use of the AH header, makes middlebox deployment difficult. However, we believe that even with widespread deployment of IPv6, NATs and proxies will continue to persist.

With NUTSS [21], Guha *et al.* focus on the problem of end-to-end addressability through NATs using SIP [28] (for signaling and long-term, stable identifiers) and STUN [29] (for NAT punching). While this approach has made substantial inroads for peer-to-peer applications such as VoIP, it is not widely used in the client/server paradigm such as the web, in which clients are largely anonymous.

NAT and proxy measurement. There are few reliable methods for detecting unique hosts behind NATs and proxies. Bellovin presents a technique using the `IPid` field in [4]. This approach relies on the host operating system using unique yet predictable `IPids`. To be effective, however, traffic from multiple clients behind the same NAT must be present within a short time period: This latter constraint makes this approach impractical for any but the most popular Internet systems. For example, fewer than 5% of the NATs sent multiple hosts to our collection site over the full 7 month measurement period. This technique is also ineffective for detecting proxies.

Another approach seeks to identify NATs themselves by examining packets for specific modifications that occur as packets traverse some types of layer-3 NATs [27]. However, this technique can not identify the number of active clients behind the NAT. Thus, such a classification is not useful for IP-based client identification: as we have shown in §4.2, while 74% of our clients are located behind NATs, the majority of NAT'd networks are only comprised of a few hosts (often just one).

Ganjam and Zhang [19] present measurements of NAT usage from a deployment of a video broadcast application. They observed environments where the percentage of NATs ranged between 35% and 80%. Their data set covered 553 hosts.

Effects of dynamic IP renumbering. IP aliasing, where a unique client is allocated multiple IP addresses via DHCP over time, has the potential to hinder a server's ability to detect unique clients. Bhagwan *et al.* [5] studied aliasing in Overnet, a peer-to-peer file sharing system. They probed 1,468 unique hosts over a period of 7 days and found that almost 40% used more than one IP address. We show that, from a server's viewpoint, aliasing does not seriously effect its ability to differentiate clients nor would result in meaningful collateral damage given IP-based blacklisting.

7 Conclusions

Conventional wisdom abounds with the drawbacks of using IP-based identification in the face of NATs, proxies, and dynamic IP renumbering. Blacklisting large proxies or NATs can result in legitimate clients losing access to desired services, while whitelisting can give access to unwanted clients. The actual extent of the problem, however, has remained largely a mystery.

Part of the challenge in uncovering the impact of edge opacity is a lack of practical techniques and deployments to "peer through the shroud" of middleboxes in the wild and at scale. In this work, we have developed, deployed, and analyzed a set of techniques that enable such measurement. Our approach combines active content—downloaded and executed by unmodified web clients—with server-side measurements.

We present seven months of measurement results from nearly 7 million clients across 214 countries. Our results show that while 74% of clients are behind NATs or proxies, most NATs are small and follow an exponential distribution. Furthermore, the few exceptional large NATs we found were easy to characterize from the server's perspective. Dynamic renumbering from DHCP generally happens on the order of days. Indeed, fewer than 2% of the clients that visited our servers over a week's period used more than two IP addresses.

Proxies, on the other hand, service client populations that may be both larger and geographically diverse. Thus, poor access control policies for proxies can have greater negative implications. However, we show that a server can detect a proxy both in real-time and using historical analysis, thus enabling a server operator to make more informed decisions.

That said, this paper's analysis is only a first step towards understanding the extent and impact of edge opacity on server decisions. We have focused on how edge opacity affects access control decisions declared over IP addresses. However, as alluded to earlier, characterizing edge opacity has implications to other domains, including regulatory compliance and fraud detection. To this end, we are currently integrating our implementation into Quova's widely-deployed IP intelligence engine [20] and plan to explore its utility in these areas.

Acknowledgments. We are very grateful to the illuminati community for contributing data by adding our web-beacons to their sites. Quova, Inc. provided access to their IP geolocation database and worked with us to develop and deploy an implementation of this work. Jeffrey Spehar contributed to our analytics website, and David Andersen, Nick Feamster, David Mazières, Dan Wendlandt, our anonymous reviewers, and our shepherd, Rebecca Isaacs, provided feedback on earlier versions of this paper.

Finally, we wish to acknowledge the support of our advisors, David Mazières and Nick McKeown.

References

[1] AOL. Transit data network. http://www.atdn.net/, 2006.

[2] H. Balakrishnan, K. Lakshminarayanan, S. Ratnasamy, S. Shenker, I. Stoica, and M. Walfish. A layered naming architecture for the internet. In *SIGCOMM*, Sept. 2004.

[3] BBC News. Bush website blocked outside US. http://news.bbc.co.uk/2/hi/technology/3958665.stm, 2004.

[4] S. M. Bellovin. A technique for counting NATted hosts. In *Internet Measurement Workshop*, Nov. 2002.

[5] R. Bhagwan, S. Savage, and G. Voelker. Understanding availability. In *IPTPS*, Feb. 2003.

[6] Bluecoat. Blue Coat Systems. http://www.bluecoat.com/, 2006.

[7] M. Casado, T. Garfinkel, W. Cui, V. Paxson, and S. Savage. Opportunistic measurement: Extracting insight from spurious traffic. In *HotNets*, Nov. 2005.

[8] D. Cheriton and M. Gritter. TRIAD: A new next generation internet architecture. http://www-dsg.stanford.edu/triad/, Mar. 2000.

[9] CoralCDN. http://www.coralcdn.org/, 2006.

[10] Cyota. http://www.rsasecurity.com/node.asp?id=3017, 2006.

[11] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Dec. 1998.

[12] Digital Envoy. http://www.digitalenvoy.net/, 2006.

[13] DIMES. http://www.netdimes.org/, 2006.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. 1999.

[15] B. Ford. Unmanaged internet protocol: Taming the edge network management crisis. In *HotNets*, Nov. 2003.

[16] P. Francis and R. Gummadi. IPNL: A NAT-extended internet architecture. In *SIGCOMM*, Aug. 2001.

[17] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, Mar 2004.

[18] M. J. Freedman, K. Lakshminarayanan, and D. Mazières. OASIS: Anycast for any service. In *NSDI*, May 2006.

[19] A. Ganjam and H. Zhang. Connectivity restrictions in overlay multicast. In *NOSSDAV*, June 2004.

[20] GeoDirectory Server. Quova, Inc. http://www.quova.com/, 2006.

[21] S. Guha, Y. Takeda, and P. Francis. NUTSS: a SIP-based approach to UDP and TCP network connectivity. In *FDNA*, Aug. 2004.

[22] HIP. Host Identity Protocol, Internet Draft, 2006.

[23] Linksys. http://www.linksys.com, 2006.

[24] G. G. Lorentz. *Bernstein Polynomials*. U. of Toronto Press, 1953.

[25] D. Meyer. University of Oregon RouteViews Project. http://www.routeviews.org/, 2005.

[26] minFraud. MaxMind, Inc. http://www.maxmind.com/, 2006.

[27] p0f v2. Passive operating system fingerprinting. http://lcamtuf.coredump.cx/p0f.shtml, 2006.

[28] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, June 2002.

[29] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - simple traversal of user datagram protocol (UDP) through network address translators (NATs). RFC 3489, Mar. 2003.

[30] Squid Web Proxy Cache. http://www.squid-cache.org/, 2006.

[31] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, Aug. 2002.

[32] L. von Ahn, M. Blum, N. Hopper, and J. Langford. CAPTCHA: Using hard AI problems for security. In *EUROCRYPT*, May 2003.

[33] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *OSDI*, Dec. 2004.

A Systematic Framework for Unearthing the Missing Links: Measurements and Impact

Yihua He, Georgos Siganos, Michalis Faloutsos, Srikanth Krishnamurthy
University of California, Riverside
yhe@cs.ucr.edu, siganos@cs.ucr.edu, michalis@cs.ucr.edu, krish@cs.ucr.edu

Abstract

The lack of an accurate representation of the Internet topology at the Autonomous System (AS) level is a limiting factor in the design, simulation, and modeling efforts in inter-domain routing protocols. In this paper, we design and implement a framework for identifying AS links that are missing from the commonly-used Internet topology snapshots. We apply our framework and show that the new links that we find change the current Internet topology model in a non-trivial way. First, in more detail, our framework provides a large-scale comprehensive synthesis of the available sources of information. We cross-validate and compare BGP routing tables, Internet Routing Registries, and traceroute data, while we extract significant new information from the less-studied Internet Exchange Points (IXPs). We identify 40% more edges and approximately 300% more peer-to-peer edges compared to commonly used data sets. Second, we identify properties of the new edges and quantify their effects on important topological properties. Given the new peer-to-peer edges, we find that for some ASes more than 50% of their paths stop going through their ISP providers assuming policy-aware routing. A surprising observation is that the degree of a node may be a poor indicator of which ASes it will peer with: the two degrees differ by a factor of four or more in 50% of the peer-to-peer links. Finally, we attempt to estimate the number of edges we may still be missing.

1 Introduction

An accurate topology model would be important for simulating, analyzing, and designing the future protocols effectively [1]. With an accurate Internet AS-level topology, first, we can design and analyze new interdomain routing protocols, such as HLP [2], that take advantage of the properties of the Internet AS-level topology. Second, we can create more accurate models for simulation purposes [3]. Third, we can analyze phenomena such as the spread of viruses [4][5], more accurately. In addition, the current initiatives of rethinking and redesigning the Internet and its operation from scratch would also benefit from such a model.

Developing an accurate representation of the Internet topology at the AS level remains as a challenge despite the recent flurry of studies [6][7][8][9][10][11][12][13]. Currently, several sources of topological information exist: (a) archives of BGP routing tables, (b) archives of BGP routing updates, (c) Internet Routing Registries, and (d) archives of traceroute data. Each of these sources has its own advantages, but each of them also provides an incomplete, sometimes inaccurate view of the Internet AS topology; these views are often complementary. Furthermore, as far as we know, IXPs (Internet Exchange Points) have not received attention in terms of Internet topology discovery, although they play a major role in the Internet connectivity.

The contributions of this work are two. (a) We design and implement a systematic framework for discovering missing links in our current Internet topology snapshot, and address two limitations of previous studies —the synthesis of different data source and incorporating topological information from IXPs. (b) We apply our framework and conduct an in-depth study of the importance of these new links, and improve our understanding of the Internet topology at the AS level. We describe our framework and our results in more detail below, while we discuss how our work complements and differs from previous efforts in the next section.

(a) A framework for identifying missing links: First, our framework identifies and validates a significant number of AS links by a careful cross-reference and synthesis of most known sources of information: BGP tables, traceroute, and IRR [14]. Second, our framework extracts significant new topological information from Internet Exchange Points (IXPs); such information is typically not used in topological studies. While prior work [15] has proposed methods to identify participating ASes at IXPs, our study greatly extends their work and overcomes certain limitations.

Note that we set a highly selective standard in our framework: we only accept edges which are verified by BGP tables or from traceroute data. In other words, we do not provide a union of the existing sources of information, but a critical synthesis. To achieve this goal, we develop a large scale traceroute-based tool, RETRO, to confirm the existence of edges, which we suspect exist.

We arrive at several interesting observations:

(i) We find a significant number of new edges: We discover *40% more edges* (15%) and *approximately 300% more peer-to-peer edges* (65%) as compared to the widely used Oregon Routeviews data set (all available BGP routing tables).

(ii) Most of the newly discovered edges are peer-to-peer edges: the current topological models have a bias by under-representing peer-to-peer edges.

(iii) Most missing peer-to-peer AS links are at the IXPs: Our results show that nearly 95% of the peer-to-peer links missed from the BGP tables are incident at IXPs. This suggests that exploring the connectivity at IXPs may help us identify hidden edges between ASes that participate at IXPs.

(iv) IRR is a good source of potential new edges: More than 80% of the new edges that are seen by considering an increased number of BGP tables were also found to exist in IRR; this indicates that IRR is a good source for finding links missing from BGP tables. Note that our IRR data is carefully filtered by the state of the art tool [16] for this purpose, which was not used by previous IRR studies.

(b) The properties and the impact of the new links: The new edges significantly change our view of the Internet AS topology. In addition, we identify interesting patterns of the new edges. Our key findings can be summarized as follows:

(i) The new edges change the models of Internet routing and financial implicatoins that previous research studies may have arrived at by using the incomplete topology models: We quantify the routing decision changes in the routing model due to the peer-to-peer edges not considered previously. We find that for some ASes, more than 50% of their paths stop going through a provider, compared to the models with incomplete AS topology. Most of these ASes are with degrees in the 10 to 300 range, *i.e.*, they are "middle-class" ASes. The financial implication of this phenomenon is that many "middle-class" ASes may not pay to their providers to the extent that was earlier expected. We conclude that business-oriented and routing studies should consider all peer-to-peer edges for accurate results.

(ii) We find that provider-customer and peer-to-peer edges have significantly different properties and they should be modeled separately: We find that the degree distribution of the provider-customer only edges can be accurately described by a power-law (with correlation coefficient higher than 99%) in all the topological instances that we examine. In contrast, degree distribution of the peer-to-peer only edges is better described by a Weibull distribution with correlation coefficient higher than 99%. These results corroborate observations made in previous studies [13][11].

(iii) The degrees of the nodes of a peer-to-peer link can vary significantly: We find that 50% of the peer-to-peer edges are between nodes whose degrees differ by a factor of more than 4 or by a degree difference of 144. This has direct implications on how we think about and model peer-to-peer edges. For instance, this observation suggests that researchers need to use caution when using the degree as an indication of whether two ASes could have a peer-to-peer relationship. Our results can provide guidelines to AS policy inference algorithms, which partly rely on the node degree.

(iv) More peer-to-peer edges may exist: We estimate that approximately 35% peer-to-peer edges, compared to the peer-to-peer edges we know at the end of this study, may still be missing. Our estimate is an educated guess on how many more possible edges we could verify, if we had more traceroute servers.

The rest of this paper is organized as follows. We review the data sources and previous work in Section 2. In Section 3, we present our framework and the motivation behind its design. In Section 4, we quantify the impact of our new found AS links. We introduce our methods to identify the IXP participants in Section 5. In Section 6, we summarize our work.

2 Background

2.1 Data Sources and Their Limitations

In this section, we describe the most popular data sources and their two main limitations: incompleteness and a bias in the nature of the discovered links.

BGP routing table dumps are probably the most widely used resource that provides information on the AS Internet topology. Each table entry contains an AS path, which corresponds to a set of AS edges. Several sites collect tables from multiple BGP routers, such as Routeview[17] and RIPE/RIS[18]. An advantage of the BGP routing tables is that their link information is considered reliable. If an AS link appears in a BGP routing table dump, it is almost certain that the link exists. However, limited number of vantage points makes it hard to discover a more complete view of the AS-level topology. A single BGP routing table has the union of "shortest" or, more accurately, preferred paths with respect to this point of observation. As a result, such a collection will not see edges that are not on the preferred path for this point of observation. Several theoretical and experimental efforts explore the limitations of such measurements [19][20]. Worse, such incompleteness may be statistically biased based on the type of the links. (Most ASes peer with each other with two types of links: the provider-customer links and peer-to-peer links. Normally, customer ASes pay their providers for traffic transit, and ASes with peer-to-peer relationship exchange traffic with no or little cost to each other.) Some types of AS links are more likely to be missing from BGP routing table dumps than other types. Specifically, peer-to-peer links are likely to be missing due to the selective exporting rules of BGP. Typically, *a peer-to-peer link can only be seen in a BGP routing table of these two peering ASes or their customers.* A recent work [13] discusses in depth this limitation.

BGP updates are used in previous studies[7][9] as a source of topological information and they show that by collecting BGP updates over a period of time, more AS links are visible. This is because as the topology changes, BGP updates provide transient and ephemeral route information. However, if the window of observation is long, an advertised link may cease to exist [7] by the time that we construct a topology snapshot. In other words, BGP updates may provide a superimposition of a number of different snapshots that existed at some point in time. Note that BGP updates are collected at the same vantage points as the BGP tables in most collection sites. Naturally, topologies derived from BGP updates share the same statistical bias per link type as from BGP routing tables: peer-to-peer links are only to be advertised to the peering ASes and their customers. This further limits the additional information that BGP updates can provide currently. On the other hand, BGP updates could be useful in revealing ephemeral backup links over long period of observation, along with erroneous BGP updates. To tell the two apart, we need highly targeted probes. Recently, active BGP probing[12] has been proposed for identifying backup AS links. This is a promising approach that could complement our work and provide the needed capability for discovering more AS links.

By using traceroute, one can explore IP paths and then translate the IP addresses to AS numbers, thus obtaining AS paths. Similar to BGP tables, the traceroute path information is considered reliable, since it represents the path that the packets actually traverse. On the other hand, a traceroute server explores the routing paths from its location towards the rest of the world, and thus, the collected data has the same limitations as BGP data in terms of completeness and link bias. One additional challenge with the traceroute data is the mapping of an IP path to an AS path. The problem is far from trivial, and it has been the focus of several recent efforts [21][22].

Internet Routing Registry (IRR)[14] is the union of a growing number of world-wide routing policy databases that use the Routing Policy Specification Language (RPSL). In principle, each AS should register routes to all its neighbors (that reflect the AS links between the AS and its neighbors) with this registry. IRR information is manually maintained and there is no stringent requirement for updating it. Therefore, without any processing, AS links derived from IRR are prone to human errors, could be outdated or incomplete. However, the up-to-date IRR entries provide a wealth of information that could not be obtained from any other source. A recent effort [16] shows that, with careful processing of the data, we can extract a non-trivial amount of correct and useful information.

2.2 Related Work and Comparison

There has been a large number of measurements studies related to topology discovery, with different goals, at different times, and using different sources of information.

Our work has the following characteristics that distinguish it from most previous other efforts, such as [13][6]: (1)We make extensive use of topological information from the Internet Exchange Points to identify more edges. It turns out that IXPs "conceal" many links which did not appear in most previous topology studies. (2)We use a more sophisticated, comprehensive and thorough tool [16] to filter the less accurate IRR data, which was not used by previous studies. (3) We employ a "guess-and-verify" approach for finding more edges by identifying potential edges and validating them through targeted traceroutes. This greatly reduced the number of traceroutes that were needed. (4)We accept new edges conservatively and only when they are confirmed by a BGP table or a traceroute. In contrast, some of the previous studies included edges from IRR without confirming it with a traceroute.

The most relevant previous work is done by Chang *et al.* [6] with data collected in 2001. They identify new edges by looking at several sources of topological information including BGP tables and IRR. They estimate that 25%-50% AS links were missing from Oregon Route-view BGP table, the most commonly used data set for AS topology studies. Their work was an excellent first step towards a more complete topology.

In a parallel effort, Cohen and Raz [13] identify missing links in the Internet topology. Our studies corroborate some of the observations there. Note that, their work does not include an exhaustive measurement, data collection and comparison effort as our work. For example, IXP information was not used in their work.

Several other interesting measurement studies exist. NetDimes [8] is an effort to collect large volumes of host-

Table 1: The topological data sets used in our study.

OBD	The Oregon routeviews BGP table dump
BD	OBD and other additional BGP table dumps
IRRnc	IRR edges processed by Nemecis with non-conflicting policy declarations
IRRdual	IRRnc edges correctly declared by both adjacent ASes
BD+IRR	BD and the edges of IRRdual confirmed by RETRO
IXPall	Union of cliques of IXP participants
ALL	BD+IRR and the potential IXP edges that are confirmed by RETRO

Table 2: The statistics of the topologies

Name	Nodes	Edges	p-c	p-p
OBD	19.8k	42.6k	36.7k	5.5k
BD	19.9k	51.3k	38.2k	12.7k
BD+IRR	19.9k	56.9k	38.2k	18.3k
ALL	19.9k	59.5k	38.2k	20.9k

Table 3: A collection of BGP table dumps

Route collector or Router server name	# of Nodes	# of Edges	# of edges with type inferred			edges not in	edges not in OBD w/ type		
			total	p-p	p-c	OBD	total	p-p	p-c
route-views(*OBD*)	19843	42643	42570	5551	36766	0	0	0	0
route-views2	19837	41274	41230	4464	36514	1029	1028	835	191
route-views.eqix	19650	34889	34876	1027	33640	674	674	530	143
route-views.linx	19655	37259	37246	3246	33765	2511	2511	2188	319
route-views.isc	19753	36152	36139	1915	34004	784	783	663	118
rrc00.ripe	19770	36479	36465	1641	34605	655	654	543	111
rrc01.ripe	19640	34193	34180	1121	32855	617	617	512	105
rrc03.ripe	19737	39147	39129	3850	35042	3233	3228	2609	616
rrc05.ripe	19765	32676	32659	1122	31324	1095	1091	658	432
rrc07.ripe	19618	32811	31797	1219	30394	804	803	724	79
rrc12.ripe	19628	33841	33827	2024	31606	1611	1610	1417	193
Total(*BD*)	19950	51345	51259	12734	38265	8702	8689	7183	1499

based traceroute information. The key here is to increase the number of traceroute points by turning cooperative end hosts into observation points. The challenge now becomes the measurement noise removal, the collection, and processing of the information [23]. Our approach and NetDimes could complement and leverage each other towards a more complete and accurate topology. Donnet *et al.* [24] propose efficient algorithms for large-scale topology discovery by traceroute probes. Rocketfuel [25] explores ISP topologies using traceroutes. In [9], the authors examine the information contained in BGP updates.

The exhaustive identification of *IXP participants* has received limited attention. Most previous work focuses on identifying the existence of IXPs. Xu *et al.* [15] develop what appears to be the first systematic method for identifying IXP participants. Inspired by their work, our approach subsumes their method, and thus, it provides more complete and accurate results (see Section 5).

3 Framework For Finding Missing Links

In this section, we present a systematic framework for extracting and synthesizing the AS level topology information from different sources. The different sources have complementary information of variable accuracy. Thus, we cannot just simply take the union of all the edges. A careful synthesis and cross-validation is required. At the same time, we are interested in identifying the properties of the missing AS links.

In a nutshell, our study arrives at three major observations regarding the properties of the missing AS links: (1) most of the missing AS edges are of the peer-to-peer type, (2) most of the missing AS edges from BGP tables appear in IRR, and (3) most of the missing AS edges are incident at IXPs. At different stages of the research, these three observations direct us to discover even more edges, some of which do not appear in any other source of information currently.

We present an overview of our work in order to provide the motivation for the different steps that we take. We start with the data set from Oregon routeviews BGP table Dump (*OBD*)[17], the BGP table dumps collected at route-views.oregon-ix.net, which is by far the most widely used data archive. Our work consists of four main steps.

A. BGP routing tables: We consider the AS edges derived from multiple BGP routing table dumps[7], and compare them to the Routeview data (OBD). The question we try to answer is what is the information that the new BGP tables bring. We use the term *BD* to refer to the combined data from all available BGP table Dumps. Table 1 lists the acronyms for our data sets.

B. IRR data: We systematically analyze the IRR data and identify topological information that seems trustworthy by Nemecis[16]. We follow a conservative approach, given that IRR may contain some out-dated and/or erroneous information. We do not accept new edges from IRR, even after our first processing, unless they are confirmed by traceroutes (using our RETRO tool). Overall, we find that IRR is a good source of missing links. For example, we discover that more than 80% of the new edges found in the new tables (*i.e.*, the AS edges in BD but not in OBD) already exist in IRR [14]. Even compared to BD, IRR has significantly more edges, which are validated by RETRO as we explain below.

C. IXPs and potential edges: We identify a set of potential IXP edges by applying our methodology on inferring IXP participants from Section 5. We find that many of the peer-to-peer edges missing from the different data sets could be IXP edges.

D. Validation using RETRO: We use our traceroute tool, RETRO, to verify potential edges from IRR and IXPs. First, we confirm the existence of many potential edges we identified in the previous steps. We find that more than 94% of the RETRO-verified AS edges in IRR indeed go through IXPs. We also discover edges that were not previously seen in either the BGP table dumps or IRR. In total, we have validated 300% more peer-to-peer links than those in the OBD data set from Route-views.

The statistics of the topologies generated from the different data sets in our study are listed in Table. 2.

3.1 The new edges from a BGP table dump

We collect multiple BGP routing table dumps from various locations in the world, and compare them with OBD. On May 12, 2005, we collected 34 BGP routing table dumps from the Oregon route collectors [17], the RIPE/RIS route collectors [18] and public route servers [26]. Several other route collectors were not operational at the time that the data was collected and therefore, we do not include them in this study. For each BGP routing table dump, we extract its "AS_PATH" field and generate an AS topology graph. We then combine these 34 graphs into a single graph and delete duplicate AS edges if any. The resulting graph, which is named as *BD (BGP Dumps)*, has 19,950 ASes and 51,345 edges that interconnect these ASes. The statistics of *BD* are similar to what was reported in [7]. Interestingly, *BD* has only 0.5% additional ASes, but 20.4% more AS edges as compared with *OBD*.

To study the business relationships of these edges, we use the PTE algorithm [27], which seems to outperform most previous such approaches. Specifically, it significantly increases the accuracy (over 90%) of inferring peer-to-peer AS links. Most of the AS edges are classified into three basic types on the basis of business relationships: provider-customer, peer-to-peer and sibling-to-sibling. Among them, sibling-to-sibling links only account for a very small (0.12%) portion of the total AS edges and we do not consider them in this study. We count the number of peer-to-peer (or "p-p" for short) and provider-customer (or "p-c" for short) AS links for each BGP routing table. The statistics for dumps with significant number of new edges are shown in Table 3.

For comparison purposes, we pick the most widely used AS graph *OBD* as our baseline graph. For each of the other BGP routing tables, we examine the number of additional AS edges that do not appear in *OBD*, as classified by their business relationship. As shown in Table 3, from each of the BGP routing tables that provides a significant number of new edges to *OBD*, most of the new-found edges are of the peer-to-peer type.

BGP table biases: underestimating the peer-to-peer edges. A closer look at the data reveals an interesting dichotomy: (1) Most edges in a BGP table are provider-customer. (2) Given a set of BGP tables, most new edges in an additional BGP table are peer-to-peer type. We can see this by plotting the types of new edges as we add the new tables. In Fig. 1, we plot the cumulative number

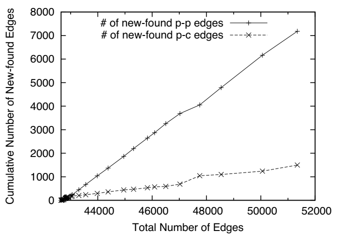


Figure 1: Most new edges in BD but not in OBD are peer-to-peer edges.

of new found peer-to-peer edges and provider-customer edges versus the total number of edges. To generate this plot, we start with *OBD* with 42643 AS edges and combine new AS edges derived from the BGP table dumps other than *OBD*, one table dump at a time, sorted by the number of new edges they provide. At the end, when all the BGP table dumps in our data set are included, we obtain the graph *BD*; this has 51345 AS edges in total. Among these edges, there are 7183 peer-to-peer edges and 1499 provider-customer edges that do not exist in the baseline graph *OBD*. Clearly, Fig 1 demonstrates that we discover more peer-to-peer AS edges than provider-customer edges when we increase the number of vantage points. Furthermore, the ratio of the number of new found peer-to-peer edges to the number of new found provider-customer edges is almost constant given that the two plots (corresponding to the new found p-p edges and the p-c edges) in Fig. 1 are almost straight lines.

The percentage of peer-to-peer edges increases with the number of BGP tables. A complementary observation is that for a BGP-table-based graph, the more complete it is (in number of edges), the higher the percentage of peer-to-peer links. For example, the AS graph derived from rrc12.ripe.net has 33841 AS edges, 2024 (5.98%) of which are peer-to-peer edges. On the other hand, the more complete AS graph *OBD* has 42643 edges, and 5551 (13.0%) of these edges are peer-to-peer edges. The combined graph *BD* has an even higher percentage (24.8%) of peer-to-peer links.

The above observations strongly suggest that in order to obtain a more complete Internet topology, one should consider peer-to-peer links than any other type of primary AS links.

3.2 Exploring IRR

We carefully process the IRR information to identify potential new edges. Recall that we do not add any edges until we verify them with RETRO later in this section.

We extract AS links from IRR on May 12, 2005 and classify their business relationships using Nemecis [16] as per the exporting policies of registered ISPs. The pur-

Table 4: AS edges in IRR (May 12, 2005) without relationship conflict

Name of Graphs	# of non-0 degree Nodes	# of Edges	Avg Degree	Perc. of total IRR edges	Perc. of IRR edges without conflict
IRRnc	16952	89540	10.56	92.6%	100.0%
peerIRRnc	6619	49411	14.93	51.1%	55.2%
pcIRRnc	15277	37619	4.925	38.9%	42.0%
siblingIRRnc	2277	2510	2.204	2.6%	2.8%
peerIRRdual	1561	18453	23.64	19.1%	20.6%
pcIRRdual	6298	8748	2.778	9.1%	9.8%
siblingIRRdual	226	143	1.265	0.1%	0.1%

pose of using Nemecis to filter the IRR is that, Nemecis can successfully eliminate most badly defined or inconsistent edges and, it can infer with fair accuracy the business relationships of the edges.

There are 96,654 AS links in total and they are classified into three basic types in terms of their relationships: peer-to-peer, customer-provider and sibling-to-sibling. Sometimes two ASes register conflicting policies with each other. For example, AS_A may register AS_B as a customer while AS_B registers AS_A as a peer. There are 7,114 or 7.4% of such AS links and we exclude them in our data analysis. We call the remaining edges *non-conflicting IRR edges* or *IRRnc*. Considering the different types of policies, this set can be decomposed into three self-explanatory sets: *pcIRRnc*, *peerIRRnc* and *siblingIRRnc*. From these edges, we define the set *IRRdual* to include the edges for which both adjacent ASes register matching relationships. (Contrarily, *IRRnc* includes edges for which only one AS registers a peering relationship while the other AS does not register at all.) Similarly, the IRRdual set can be decomposed by type of edge into three sets: *pcIRRdual*, *peerIRRdual* and *siblingIRRdual*.

The statistics of these data sets are summarized in Table 4. We notice that the number of edges in the more reliably defined *IRRdual* set is significantly less than that of the *IRRnc*. In other words, AS edges in *IRRdual* and its subsets (*peerIRRdual*, *pcIRRdual* and *siblingIRRdual*) are fewer but we are more confident about: (a) their existence, and (b) their business relationships.

We make the following two observations:

a. IRR is a good source of hints for missing edges. We perform the following thought experiment: *knowing only the OBD data set, would IRR be a good source of potential edges?* We compare the edges in graph *BD* but not in graph *OBD* with the edges in IRR. We find that 83.3% of these edges exist in IRR: 7251 from a total of 8702 new edges. This high percentage suggests that the IRR can potentially be a source for finding new edges. We also notice that from among these 7251 edges, 6302 are classified in terms of their business relationships by Nemecis[16]. From among these classified edges, 5303 edges are of the peer-to-peer type and only 832 are of the provider-customer type. This confirms the result shown in Fig. 1, where most new found AS edges are of the peer-to-peer type. Recall that, for Fig. 1, the business relationships are inferred by the PTE algorithm[27], instead of Nemecis[16], which we use here. Both algorithms give quantitatively similar results which provides high credibility to both the data and the interpretations.

Table 5: Percentage of IRR edges missing from *BD*

Name	# of edges	# of edges NOT in *BD*	# of edges Missing Perc.
IRRnc	89,540	63,660	71.1%
peerIRRnc	49,411	39,894	80.7%
pcIRRnc	37,619	22,466	59.7%
siblingIRRnc	2,510	1,300	51.8%

b. IRR has many more edges compared to our most complete BGP-table graph (BD). Motivated by the observation above, we examine the number of AS edges in IRR that are not included in *BD*. Table 5 summarizes the number and the type of IRR AS edges that do not appear in *BD*. From among the IRR AS edges inferred as non-conflicting types, 71.1% are missing from *BD*. The percentage is especially high for peer-to-peer edges: 80.7% of the peer-to-peer AS edges in IRR are missing from *BD*. This suggests that there may be many IRR links that exist but are yet to be verified. We also notice that 59.7% of the provider-customer AS edges are missing. At this point, we can only speculate that most of these missing provider-customer AS edges represent backup links.

3.3 IXPs and missing links

Note that, when two ASes are participants at the same IXP, it does not necessarily mean that there is an AS edge between them. If two participating ASes agree to exchange traffic through an IXP, this constitutes an AS edge, which we call an *IXP edge*. Many IXP edges are of peer-to-peer type, although customer-provider edges are also established.

Identifying IXP edges requires two steps: (a) we need to find the IXP participants, and (b) we need to identify which edges exist between the participants. We defer a discussion of our method and tool on how to find the IXP participants to Section 5. However, even when we know the IXP participants, identifying the edges is still a challenge: not all participants connect with each other. In addition, the peering agreements among the IXP participants are not publicly known.

We start with a superset of the real IXP edges that contains all possible IXP edges: we initially assume that the participants of each IXP form a clique. We denote by *IXPall* the set of all edges that make up all of these cliques.

Table 6: Many missing peer-to-peer links are at IXPs

Name	# of Edges	$\bigcap$ *IXPall*	Perc.
peerBD-OBD	7183	6197	86%
peerIRRnc-BD	39894	23979	60%
peerIRRdual-BD	13905	11477	83%
BD-OBD	8702	6910	79%

IXPall contains 141,865 distinct AS edges.

Potential missing edges and IXP edges. We revisit the previous sets of edges we have identified and check to see if they could be IXP edges. First, we look at the peer-to-peer AS edges that appear in *BD* but not in *OBD*. These are the peer-to-peer AS edges missing from *OBD* but are discovered with *BD*. We call this set of AS edges *peerBD-OBD*. Here we use the minus sign to denote the difference between two sets: *A-B* is the set of entities in set *A* but not in set *B*. Second, we look at the AS edges that appear in *peerIRRnc* but not in the graph *BD*. We call this set of links *peerIRRnc-BD*. These AS links are the ones that are potentially missing from *BD*. We define the *peerIRRdual* links not in *BD* as *peerIRRdual-BD*.

Having made this classification, we compare each class with the super set, *IXPall*, of edges that we constructed earlier. The statistics are shown in Table 6. With our first comparison, we find that approximately 86% of the edges in *peerBD-OBD* are in *IXPall* and hence, are potentially IXP edges. Next, we observe that 60% of the edges in *peerIRRnc-BD* and 83% of the edges in *peerIRRdual-BD* are in *IXPall*. Thus, if they exist, they could be IXP edges.

In summary, the analysis here seems to suggest that, most of the peer-to-peer AS links missing from the BGP dumps but present in IRR are potentially IXP edges.

3.4 Validating links with RETRO

With the work so far, we have identified sets of edges and obtained hints on where to look for new edges: (1) most missing links are expected to be the peer-to-peer type, (2) IRR seems to be a good source of information, (3) many missing edges are expected to be IXP edges.

However, as we have noted before, the peer-to-peer edges learned through the IRRs and *IXPall* are not guaranteed to exist. Therefore, in this section we focus on validating their existence to the extent possible. *Note here that with the validation, we eliminate stale information that may still be present in the IRR and IXP data sources.*

To verify the existence of the edges in *peerIRRnc-BD*, we would like to witness these edges on traceroute paths. Typically, when a traceroute probe passes through an IXP edge between AS A and AS B, it will contain the following sequence of IP addresses: $[IP_{AS\_A}, IP_{IXP}, IP_{AS\_B}]$. If such a pattern is observed with our traceroute probes, it is almost certain that an IXP edge between AS A and AS B exists.

We first tried to use the Skitter[28] traces as our verification source; however, we soon found that it was not suitable for our purposes. Between May 8 and May 12 in 2005, we collected a full cycle of traces from each of the active Skitter monitors. Despite a total number of 21,363,562 individual traceroute probes in the data set, we were only able to confirm 399 IXP edges in *peerIRRnc-BD*. The reason could be that the monitors were not in the "right" place to discover these edges: the monitors should be at the AS adjacent to that edge, or at one of the customers of those two ASes. With the limited number of monitors (approximately two dozen active ones) in Skitter, it is difficult to witness and validate many of the peer-to-peer AS edges.

To address this limitation, we develop a tool for detecting and verifying AS edges. We employ public traceroute servers(*e.g.*[29]) to construct RETRO (REverse TraceROute), a tool that collects traceroute server configurations, send out traceroute requests, and collect traceroute results dynamically. Currently, we have a total of 404 reverse traceroute servers which contain more than 1200 distinct and working vantage points. These vantages points cover 348 different ASes and 55 different countries.

With the RETRO tool, we conduct the following procedure to verify AS edges in the *peerIRRnc-BD* set. For each edge in *peerIRRnc-BD*, we find out if there are any RETRO monitors in at least one of the two ASes incident on the edge. For about 2/3 of the edges in *peerIRRnc-BD*, we do not have a monitor in either of the two ASes on the edge. If there is at least one monitor, we try to traceroute from that monitor to an IP that belongs to the other AS on the edge. There are two problems in finding the right IP address to traceroute to. First, some ASes do not announce or can not be associated with any IP prefixes and thus, we are not able to traceroute to these ASes. Second, most of the rest of the ASes announce a large range (equal to or more than 256, *i.e.*, a full /24 block) of IP addresses. To maximize our chances of performing a successful traceroute, we choose a destination from the list of IP addresses that has been shown to be reachable by at least one of the Skitter monitors. We then trigger RETRO to generate a traceroute from the selected monitor to the destination IP address that we choose. We call this set of traceroutes *RETRO_TRACE1*.

Most missing peer-to-peer links are incident at IXPs. We define a *candidate* to be a potential edge between two ASes, which satisfy the following two conditions: (a) we have a RETRO monitor located in one of the two ASes, and (b) there is at least one IP address from the other AS is reachable by the traceroute probe performed from the RETRO monitor. We have 8791 such "candi-

Table 7: RETRO verifies peer-to-peer links in IRR missing from BD

Name	# of edges	# of RETRO candidates	# of confirmed peering		
			total	via IXP	direct
peerIRRnc-BD	39894	8791	5646	5317	329
peerIRRdual-BD	13905	4487	3529	3351	178

Table 8: RETRO verifies AS edges not in *BD* and *IRRnc*

Name	# of edges	# of RETRO candidates	# of confirmed peering		
			total	via IXP	direct
IXPall-BD-IRR	100,076	17,640	2,603	2,407	196

dates" for the potential AS edges in *peerIRRnc-BD*. By appropriately performing traceroutes on candidates, we get traceroute paths. In these paths, we search for two patterns for each candidate (AS_A, AS_B): (a) [IP_{AS_A}, IP_{AS_B}]. , and (b) [IP_{AS_A}, IP_{IXP}, IP_{AS_B}]. If either of the two patterns appears, it is almost certain that the AS edge between AS_A and AS_B exists either as (a) a direct edge or, (b) as an IXP edge, respectively. The results that we obtain at the end of the above process are summarized in Table 7.

Among 8791 candidates in *peerIRRnc-BD*, RETRO is able to confirm that a total of 5646 edges indeed exist. The existence of the rest of the candidates does not show in our RETRO data. Note that this method can only confirm the presence, but not prove the absence of an edge. It could very well be that that the traceroute does not pass through the right path. The most interesting result is, from among the 5646 verified edges, 5317 or 94.2% of them are IXP edges. The result suggests that most of the missing peer-to-peer links from BGP tables are in fact incident at IXPs. We conjecture that this is probably because the peer-to-peer links between middle or low ranked ASes (national or regional ISPs) are typically underrepresented in BGP tables. For those ASes, peering with other ASes at IXPs is a much more cost-efficient way than by building private peering links one by one. Our result strongly suggests that in order to look for missing peer-to-peer links from BGP tables, we should examine IXPs more carefully.

Discover edges not observed in BGP tables or IRRs From the results so far, we suspect that the missing edges are often IXP edges. Following this pattern, we identify and confirm edges that previously had not been observed in any other data source.

We consider those AS edges in *IXPall* that are neither in *BD* nor in *IRRnc*, and call them *IXPall-BD-IRR*. We then attempt to trace these edges by using RETRO. We call this set of traceroute *RETRO_TRACE2*. The results from our experiments are summarized in Table 8.

We find 2,603 new AS edges from out of 17,640

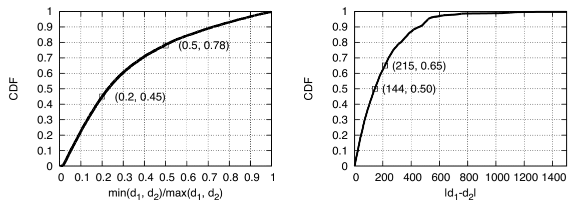


Figure 2: Degree ratio distribution(left) and degree difference distribution (right) of all peer-to-peer AS links in the Internet.

RETRO candidate paths. The percentage of confirmed new AS edges is 14.8%. This is much lower than what we see with *peerIRRnc-BD*. This is due to the fact that IXPall is an overly aggressive estimate. In addition, we have already identified that many edges from IXPall are in the previous sets (*BD* and *peerIRRnc-BD*).

We also notice that there is a small number of confirmed edges that are shown to exhibit direct peering instead of peering at some IXP. A closer look reveals that many of such cases are due to the fact that a small number of routers do not respond with ICMP messages with the incoming interfaces, and therefore, the IXP IP address, which is supposed to be returned by the traceroute, is "skipped". Note that this phenomenon does not stop us from identifying the edge. It just makes us underestimate the percentage of IXP edges among the confirmed edges.

4 Significance of the new edges

In this section, we identify properties of the new edges. Then, we examine the impact of the new edges on the topological properties of the Internet. Finally, we attempt to extrapolate and estimate how many edges we may still be missing.

4.1 Patterns of the peer-to-peer edges

We study the properties exhibited by nodes that peer. Therefore, we examine the degrees, d_1 and d_2, of the two peering nodes that make up each peer-to-peer edge. Let us clarify that the degrees d_1 and d_2 include both peer-to-peer and provider-customer edges. One would expect that d_1 and d_2 would be "comparable". Intuitively, one would expect that the degree of an AS is *loosely* related to the importance and its place in the AS hierarchy; we expect ASes to peer with ASes at the same level.

However, we find that *the node degree of the nodes connected with a peer-to-peer link can differ significantly.* We compare the two degrees using their ratio and absolute difference. Note that these two metrics provide complementary view of difference, which leads to the following two findings: (1) Close to 78% of the peer-to-peer

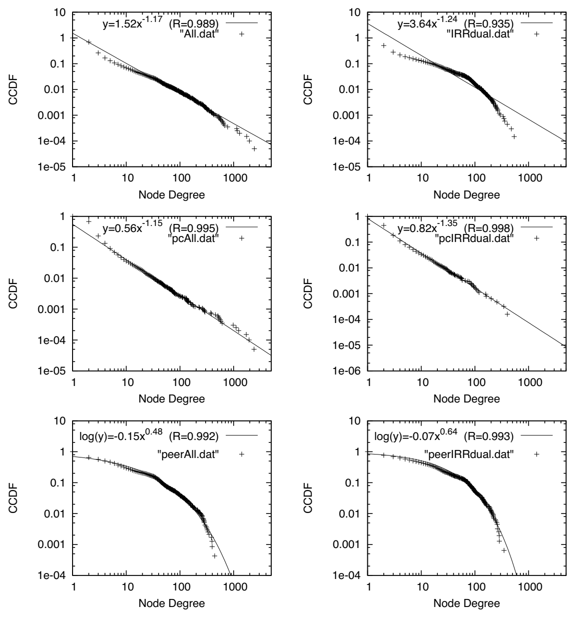


Figure 3: The degree distributions of *ALL* (left) and *IRRdual* (right) in the top row, their provider-customer degree distributions in the middle row, and their peer-to-peer degree distributions in the bottom row.

edges connect ASes whose degrees differ by a factor of 2. In Fig. 2 (left), we plot the CDF of the distribution of the ratio $min(d_1, d_2)/max(d_1, d_2)$ of the peer-to-peer edges. Another observation is that 45% of the peer-to-peer edges connect nodes whose degrees differ by a factor of 5. This is a surprisingly large difference. One might argue that this is an artifact of having peer-to-peer edges between low degree nodes, say $d_1 = 2$ and $d_2 = 11$, whose absolute degree difference is arguably small. This is why we examine the absolute difference of the degrees next. (2) 35% of the peer-to-peer edges have nodes with an absolute difference greater than 215. In Fig. 2 (right), we plot the CDF of the distribution of the absolute value $|d_1 - d_2|$, where d_1 and d_2 remain as defined earlier. Another interesting observation is that approximately half of the peer-to-peer edges have a degree difference larger than 144. Differences of 144 and 215 are fairly large if we consider that roughly 70% of the nodes have a degree less than 4. We intend to investigate why quite a few high degree ASes establish peer relationship with low degree ASes in the future.

4.2 Impact on the Internet topology

4.2.1 The degree distribution

There has been a long debate on whether the degree distribution of the Internet at the AS level follows a power-law[30][31][32][6]. This debate is partly due to the absence of a definitive statistical test. For example, in Fig. 3 top left, we plot the complementary cumulative distribution functions (CCDF), on a log-log scale, of the graph *ALL* defined earlier in Table 1. The distribution is highly skewed, and the correlation coefficient of a least square errors fitting is 98.9%. However, one could still use different statistical metrics and argue against the accuracy of the approximation [32].

Furthermore, the answer could vary depending on which source we think is more complete and accurate, and the purpose or the required level of statistical confidence of a study. For example, if we go with *IRRdual*, which is a subset of the AS edges recorded in IRR filtered by Nemecis, the correlation coefficient is only 93.5%, see Fig. 3 top right.

To settle the debate, we propose a reconciliatory divide-and-conquer approach. We propose to model separately the degree distribution according to the type of the edges: provider-customer and peer-to-peer. We argue that this would be a more constructive approach for modeling purposes. This decomposition seems to echo the distinct properties of the two edge types, as discussed in a recent study of the evolution on the Internet topology [11].

In Fig. 3, we show an indicative set of degree distribution plots for graph *ALL* on the left column and *IRRdual* on the right. We show the distributions for the whole graph (top row), the provide-customer edges only (middle row), and the peer-to-peer edges only (bottom row). We display the power-law approximation in the first two rows of plots and the Weibull approximation in the bottom row of plots.

We observe the following two properties: (a)The provider-customer-only degree distribution can be accurately approximated by a power-law. The correlation coefficient is 99.5% or higher in the plots of Fig.3 in the middle row. Note that, although the combined degree distribution of *IRRdual* does not follow a power law (top row right), its provider-customer subgraph follows a strict power law (middle row right). (b)The peer-to-peer-only degree distribution can be accurately approximated by a Weibull distribution. The correlation coefficient is 99.2% or higher in the plots of Fig.3 in the bottom row.

It is natural to ask why the two distributions differ. We suggest the following explanation. Power-laws are related to the rich-get-richer behavior: low degree nodes "want" to connect to high degree nodes. For provider-customer edges, this makes sense: an AS wants to connect to a high-degree provider, since that provider would likely provide shorter paths to other ASes. This is less boviously rue for peer-to-peer edges. If AS1 becomes a peer of AS2, AS1 does not benefit from the other peer-to-peer edges of AS2: a peer will not transit traffic for a peer. Therefore, high peer-to-peer degree does not make

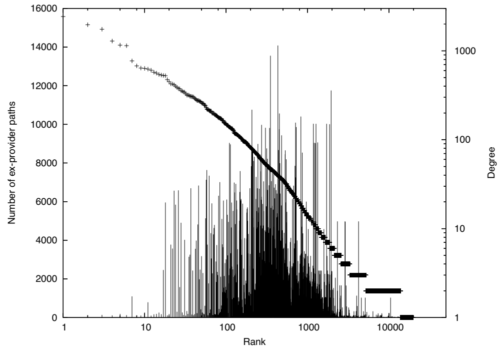


Figure 4: The number of *ex-provider paths* (shown as impulses on the left y-axis) of each node in order decreasing node degree (shown as a semi diagonal line corresponding to the right y-axis). The x-axis shows the rank of the nodes in the order of descending degree.

a node more attractive as a peer-to-peer neighbor. We intend to investigate its validity in the future.

4.2.2 Clustering coefficient

We expect that the *ALL* graph will be more clustered since we add edges. To quantify this, we use the *clustering coefficient* which has been used to characterize and compare generated and real topologies [33]. Intuitively, the clustering coefficient captures the extent to which a node's one-hop neighborhood is tightly connected. A clustering coefficient of exactly one means that the neighborhood is a clique. The average clustering coefficient of *OBD* is 0.25 and it increases to 0.31 in *ALL*.

In addition, we find that the density increase is not homogeneous. The neighborhoods of "middle-class" nodes become more clustered: the clustering coefficient increase is larger for nodes with degrees in the 10 to 300 range. Note that this property characterizes the new edges, and could help us identify more missing edges in future studies.

4.2.3 AS path length

We study the effect of the new edges on the AS path lengths with policy-aware routing. The routing policy is a consequence of the business practices driven by contracts, agreements, and ultimately profit. As a first-order approximation of the real routing policy, we use the *No Valley Prefer Customer (NVPC)* routing, which is defined in [34] [35].

We have approximately 20,000 ASes present in the Internet topology and examine all possible pairs of ASes. For each AS pair, we compare the AS path lengths with *OBD* and with *ALL*. We find that approximately 10 million of the paths change in length. While we note that this is a small fraction of the total number of paths, it is still a significant number in terms of its absolute value. In addition, *no change in the length does not mean that the path did not change*. For this reason, we study next how many paths changed even if they did not change in length.

4.3 The effect on ISP revenue

We examine how much the new discovered AS links would change the models previous studies had arrived at about routing decisions and ISP income by using incomplete Internet topology.

Similar to studying AS path length, we assume NVPC routing in our model. For each AS, we count how many of its paths stop going through one of its providers once the new edges are added. We refer to these paths as *ex-provider paths*. The number of ex-provider paths is an indication, of the financial gains for that AS. Clearly, there are other considerations, such as prefix-based traffic engineering and performance issues, that our analysis cannot possibly capture. However, our results are a good first indication of the effect of the new peer-to-peer links.

The significant financial benefits of the new peer-to-peer edges. We plot the number of the *ex-provider* paths for each node in Fig. 4. The x-axis represents the rank of the nodes on a log scale in order of decreasing degree; The y-axis at the left represents the number of ex-provider paths. In addition, we plot the node degrees (on the right y-axis) against their ranks as a semi diagonal line. We see that the difference between using an incomplete graph (*OBD*) and using a more complete graph (*ALL*) is dramatic: there are many ASes, for each of which, several thousands out of the total 20K paths (to all other ASes) stop going through a provider. For some ASes, more than 50% of their paths stop going through their providers (10K out of 20K possible paths per AS).

The rise of the "middle class" ASes. Another interesting observation is that the nodes which seem to benefit the most from these changes have degrees in the range from 10 to 300 (right y-axis). Top tier nodes (top 20 ranked) almost do not benefit at all; this is expected, since they do not have any providers anyway. Nodes with really low node degree do not benefit much either, since nodes with very low degrees are less likely to have a peer-to-peer edge.

4.4 Are we missing a lot more peer edges?

Currently, the *ALL* graph has approximately 20.9K peer-to-peer edges. However, we were very conservative in adding edges from *IRRnc*: we required that the edges are verified by RETRO. So, a natural question is, how many more edges could we verify from *IRRnc* if we had more

RETRO servers? We attempt to provide an estimate by extrapolating the success of our method in finding new edges. First, we provide a conservative estimation and later, a more liberal estimation, below.

Conservative extrapolation using IRRdual: We find 35% more peer-to-peer edges compared to ALL. We revisit the *IRRdual* graph and examine if we can include more edges than the ones we validate with RETRO. Recall from Table 7 that we find that there are 13905 edges in the *peerIRRdual-BD*, and from these, only 4487 are "verifiable" candidates. From the verifiable edges, we actually verify 3529 or 78.6% of the verifiable edges. We generalize this percentage: we assume that if we had more RETRO monitors, we could verify 78.6% of the *peerIRRdual-BD*. This leads to an estimated 7.4K (10.9K−3.5K) peer-to-peer edges not in ALL, which has 20.9K peer-to-peer edges.

Liberal Extrapolation using IRRnc: We find 95% more peer-to-peer edges compared to ALL. In a similar way, we estimate how many edges we could verify from *peerIRRnc-BD*, which is a more "inclusive" set. Here, the total number of peer-to-peer edges is 39,894, the verifiable edges 8,791, and the verified edges 5,646. This gives rise to an estimate of $39894 \times 5646/8791 = 25.6$K peer-to-peer edges out of which 5.6K are already in ALL.

5 Identifying IXP Participants

In this section, we present a method for identifying the *participants* at Internet Exchange Points (IXPs). Our goal is to find all the participants at each IXP, and this is a non trivial problem. We find that finding the IXP participants is key for identifying many missing AS edges as explained in section 3.

5.1 From IPs to IXP participants

This part of our approach uses two techniques to infer IXP participants from IXP IP addresses: 1)path-based inference, where we perform a careful processing of collected traceroute data, and 2)name-based inference, where, we analyze the name and the related information with regard to IXPs from the DNS and/or WHOIS databases.

In both inference methods, we start with the IP address blocks allocated to the IXPs, which we call *IXP IP addresses*. We obtain this information from the Packet Clearing House (PCH) [36]. In terms of traceroute data, we use a full cycle of Skitter traceroute data between May 1, 2005 and May 12, 2005, and our *RETRO_TRACE1* data in May 2005 as described in Section 3.4.

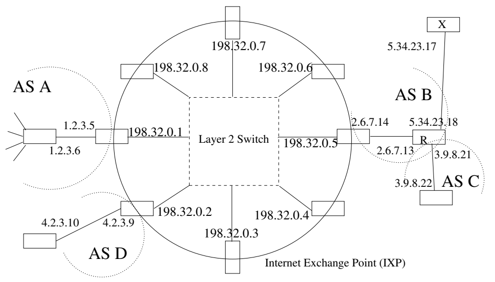


Figure 5: A conceptual model of a typical IXP

5.1.1 Path-based inference

The high level overview of the method is deceptively simple. First, for each IXP IP address IP_{ixp} that we obtain from PCH, we search for the IP address that appears immediately after IP_{ixp} in each of the obtained traceroute paths. Second, if we find more than one such IP address for the particular IP_{ixp}, we select the one that appears most frequently to be IP_{next}. We call the above procedure the *majority selection process*. Third, we find the AS ASx that owns the IP address, IP_{next} and consider that ASx to be a participant at the IXP. Furthermore, we consider that IP_{ixp} is the IP interface via which ASx accesses the IXP.

To illustrate this with an example let us consider Fig. 5. A typical traceroute from AS A to router X yields the following sequence of IP addresses: [1.2.3.5, 198.32.0.5, 2.6.7.13, 5.34.23.17]. Since the address "2.6.7.13", which belongs to AS B, appears immediately after IXP IP address "198.32.0.5", we infer that, AS B is a participant AS, and that 198.32.0.5 is the interface that is assigned to AS B. Note from Fig. 5 that, irrespective of the location of the traceroute source and its destination, if an IXP address (the address 198.32.0.5 in our example) appears in a traceroute, the IP address that appears immediately after (the address 2.6.7.13 in our example) is owned by the AS (in our example AS B) that uses the IXP address (*e.g.* 198.32.0.5) to access the IXP as long as two conditions hold. These are: (1) each IXP interface address is assigned to a single AS, and (2) routers *always* respond to a traceroute probe with the address that corresponds to the incoming IP interface. While the first condition largely holds, the second condition does not. There is a small chance that a router could respond to a traceroute probe with an alternate (not the incoming) interface[37][21]. In our example, router R could respond to a traceroute probe from AS A to router X with an alternate interface (*e.g.* 3.9.8.21), which makes the traceroute path appear as [1.2.3.5, 198.32.0.5, 3.9.8.21, 5.34.23.17]. Since 3.9.8.21 could be within the IP space of AS C, one could incorrectly infer that AS C is an IXP participant. We overcome this limitation with our *majority-selection process*; the

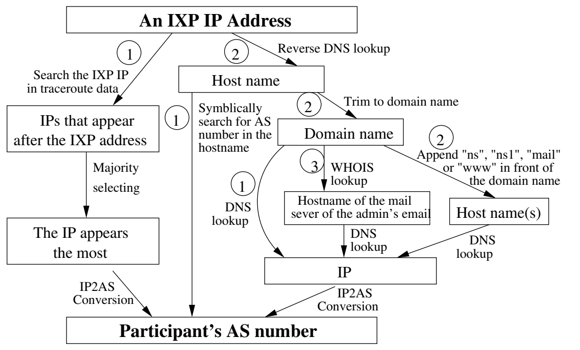


Figure 6: A flow chart of our path-based method to infer IXP participants from IXP IP addresses. Starting from the top, the numbers in the circle indicate the priority (lower number with higher priority) at a branching point.

Table 9: IXP participants inferring comparison

Name of IXP	Actual partici-pants	XDZC Approach [15]				Our Approach			
		correctly inferred	total inferred	$\mathcal{R}$	$\mathcal{P}$	correctly inferred	total inferred	$\mathcal{R}$	$\mathcal{P}$
MSK-IX	**154**	**90**	115	68%	90%	**136**	156	88%	87%
JPIX	**110**	**58**	82	53%	71%	**107**	128	97%	84%
FREEIX	**101**	**38**	39	38%	97%	**64**	65	63%	98%
AMS-IX	**211**	**177**	220	84%	80%	**182**	200	86%	91%
LINX	**175**	**164**	242	94%	68%	**168**	193	96%	87%
DE-CIX	**144**	**111**	124	77%	90%	**137**	142	95%	96%

basis is the assumption that *in the majority of the cases*, routers will respond to a traceroute probe with the incoming interface. This assumption has been shown to hold by numerous prior efforts [37][21].

The previously proposed method in [15] does not have the majority selection process. Furthermore the method does not associate the specific IXP IP interface addresses with their respective participating ASes. Our majority selection process eliminates measurement noise and thus, ensures a lower "false positive" rate. We map the discovered AS participants to their assigned IXP IP addresses, and using this, exclude the addresses in the name-based inference process that we describe below. This practice reduces the number of total IXP IP addresses that are subject to the name-based inference procedures which are inherently less reliable, and thus reduces the possible errors overall.

5.1.2 Named-based IXP participants inference.

The basic name-based IXP participants inference method, which was proposed in [15], works in three main steps: (a) for every IP address in each IXP prefix space, we do a reverse DNS look up, and we find the host name for that IXP IP address, (b) we take the domain name part (*company.*{*com,net,org, etc.*}) from the host name, and do a DNS look up, which leads to a new IP address, and (c) we find the AS that owns this address, and this AS is considered a participant of that IXP. For example, IXP DE-CIX has the IP address 80.81.192.186. If we do a reverse DNS lookup, we get the host name "GigabitEthernet3-2.core1.ftf1.level3.net". A DNS lookup of the domain name "level3.net" yields an IP address of 209.245.19.41. An IP address to AS number conversion reveals that the IP address belongs to AS3356 (Level3). Therefore, AS3356 is considered a participant at DE-CIX.

Although this method has been used successfully by previous studies [15], it has two limitations: (a) sometimes it can return incorrect AS numbers for IXP participants, and (b) it does not always work: the DNS or the reverse DNS lookup may not return an answer.

We address the first limitation by excluding the IXP addresses that have been mapped on to AS participants by our path-based inference method. This greatly reduces the number of IXP addresses that are to be examined by the named-based inference method and therefore reduces the possible number of erroneous results.

We address the second limitation by proposing three new methods to improve the success rate of name-based inference:

a. Examining host names containing AS numbers. Sometimes, the DNS name of an IXP IP address contains the AS number of an IXP participant. For example, 195.66.224.71 is an IP address at the London Internet Exchange (LINX), which has a DNS name fe-3-4-cr2.sov.as9153.net. From that, we can infer that AS9153 is a participant at the LINX IXP.

b. Examining common naming practices. We can increase the success rate of DNS lookups by including common host names with the inferred domain names. For example, although *company.net* may fail to be resolved, the DNS look up may succeed with *ns.company.net*. In fact, there are several common host names such as "ns", "ns1", "mail" and "www". Hosts with these names *usually* belong to the same AS. For example, 195.66.226.104 is an IP address at IXP LINX at London, England. The host name of that IP address is "linx-gw4.vbc.net" and the DNS lookup for the domain name "vbc.net" is unsuccessful. However, the DNS lookup for ns.vbc.net returns the address 194.207.0.129, which belongs to AS8785 (Astra/Eu-X and VBCnet GB).

c. Using the administrating personnel information. A WHOIS lookup for a domain name often has an administrative/technical contact person's e-mail address. The mail server is often within the same AS that corresponds to the domain name. For example, for "decix-gw.f.de.bcc-ip.net", all DNS lookups described previously, fail. However, if we look at the WHOIS lookup for domain "bcc-ip.net", we will find the contact email server is "bcc.de", which has an IP address of 212.68.64.114, and it belongs to AS9066 (BCC GmbH).

5.1.3 Putting the two techniques together

We integrate both the path-based and named-based techniques, into a tool for inferring IXP participants from IXP addresses. We start with the path-based technique, and for every IP address in the IP block of an IXP, we try to find it in a traceroute path. If this works, then we do not reexamine this IP address. Otherwise, we use the name-based inference and we utilize the three mechanisms that we proposed above. For completeness, we show the flow chart of the inference method in Fig. 6.

5.1.4 Evaluating our inference approach

We use two complementary metrics: *Recall* $\mathcal{R}$ and *Precision* $\mathcal{P}$, which are widely used in the data mining literature for similar tasks. They are defined as follows: $\mathcal{R} = \frac{\mathcal{N}_{correct}}{\mathcal{N}_{actual}}$ and $\mathcal{P} = \frac{\mathcal{N}_{correct}}{\mathcal{N}_{inferred}}$ where $\mathcal{N}_{correct}$ is the number of correctly inferred participants from among those inferred, $\mathcal{N}_{actual}$ is the actual number of participants, and $\mathcal{N}_{inferred}$ is the total number of inferred participants. Note that the Precision metric, $\mathcal{P}$, has not been used in previous studies although it is critical for detecting false positives. Otherwise, we favor overly aggressive inference methods that suggest a large number of correct and incorrect participants.

For the comparison and for lack of a better criterion, we select the six largest IXPs (in terms of number of participants) for which we know the participants through the EURO-IX site [38] or the IXPs' own web sites. In Table 9, for each IXP, we list its actual number of participants, the number of ASes that our algorithm inferred, and the number of ASes that our algorithm inferred *correctly*. We also show the Recall and Precision metrics.

It is easy to see that: (a) our approach is very effective in determining most of the participants in these IXPs, and (b) our approach identifies correctly more participants than XDZC[15] and almost always with better Precision. For the case of MSK-IX, we only have slightly lower Precision (by 3%) but a significantly higher Recall (by 20%).

5.2 From web-based archive

We notice there are some limitations on inferring IXP participants by the IXP IP addresses alone. For example, some IXPs do not have globally routable IP addresses and some IP addresses are either invisible by traceroute or appear as "*"s in responses to traceroute probes.

To overcome these limitations, we include an additional source of information by retrieving IXP participant information from the web sites. We have developed a tool that automatically downloads and parses the web pages, and outputs the AS numbers of the participants periodically. We use the European Internet Exchanges Association [38] which maintains a database with 35 IXPs and their participants. We are also able to collect information from the web pages of 31 other IXPs. Naturally, as any manually-maintained data, these archives can also contain inaccuracies. However, we did not find any major inconsistencies with our measured data.

5.3 The combined results

We applied our methods to infer the participants at various IXPs on May 12, 2005. We first use our web-based archival inference. For the rest of the IXPs, we collect information with regard to their IP address blocks from Packet Clearing House [36], and infer their participants from their IXP IP addresses by using our inferring heuristics. We identify 2348 distinct participants at 110 IXPs. Some ASes actively participate in multiple IXPs. For example, AS 8220 (Colt Telecom) is inferred as a participant in 22 different IXPs in 15 different countries. In this study, we have used the combined results as our source of IXP data.

6 Conclusion

In a nutshell, our work develops a systematic framework for the cross-validation and the synthesis of most available sources of topological information. We are able to find and *confirm* approximately 300% additional edges. Furthermore, we recognize that Internet Exchange Points (IXPs) hide significant topology information and most of those new discovered peer-to-peer AS links are incident at IXPs. The reason for such a phenomenon is probably because, most missing peer-to-peer links are likely to be at the middle or lower level of the Internet hierarchy, and peering at some IXP is a cost-efficient way for the ASes to setup peering relationships with other ASes. We show that by adding these new AS links, some research results based on previous incomplete topology, such as routing decision and ISP profit/cost, change dramatically. Our study suggest that business-oriented studies of the Internet should make a point of taking into consideration as many peer-to-peer edges as possible.

So, how many AS links are still missing from our new snapshot of the Internet topology? Our findings suggest that if we know the peering matrix of all the IXPs, we might be able to discover most of the missing peer-to-peer AS links. Unfortunately, very few IXPs publish their peering matrices. Futhermore, the published peering matrices are not necessarily accurate, complete or up-to-date. In our conservative estimates, there might be still 35% hiding peer-to-peer edges, in addition to what we already have in current Internet AS graph.

Our future plans have two distinct directions. First, we want to continue the effort towards a more complete Internet topology instance. Using the framework we developed here, we are in a good position to quickly and accurately incorporate new information, such as new BGP routing tables, or new traceroute servers. Second, given our more complete AS topology, we are in a better position to understand the structure of the Internet and the socio-economic and operational factors that guide its growth. This in turn could help us interpret and anticipate the Internet evolution and, indirectly, give us guidelines for designing better networks in the future.

References

[1] S. Floyd and V. Paxson. Difficulties in simulating the Internet. *IEEE Transaction on Networking*, Aug 2001.

[2] L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica. HLP: A Next-generation Interdomain Routing Protocol. In *ACM Sigcomm*, 2005.

[3] O. Maennel and A. Feldmann. Realistic BGP Traffic for Test Labs. In *ACM Sigcomm*, 2002.

[4] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law Internets. In *ACM Sigcomm*, Aug 2001.

[5] A. Ganesh, L. Massoulie, and D. Towsley. The Effect of Network Topology on the Spread of Epidemics. In *IEEE infocom*, 2005.

[6] H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. Towards capturing representative AS-level Internet topologies. *Computer Networks*, 44(6):737–755, 2004.

[7] B. Zhang, R. Liu, D. Massey, and L. Zhang. Collecting the Internet AS-level Topology. *ACM SIGCOMM Computer Communication Review(CCR)*, January 2005.

[8] Y. Shavitt and E. Shir. DIMES: Let the Internet Measure Itself. *ACM SIGCOMM Computer Communication Review (CCR)*, October 2005.

[9] X. Dimitropoulos, D. Krioukov, and G. Riley. Revisiting Internet AS-Level Topology Discovery. In *PAM*, 2005.

[10] P. Mahadevan, D. Krioukov, M. Fomenkov, B. Huffaker, X. Dimitropoulos, kc claffy, and A. Vahdat. The Internet AS-Level Topology: Three Data Sources and One Definitive Metric. *ACM SIGCOMM Computer Communication Review (CCR)*, January 2006.

[11] H. Chang, S. Jamin, and W. Willinger. To Peer or not to Peer: Modeling the Evolution of the Internet's AS Topology. In *IEEE Infocom*, 2006.

[12] L. Colitti, G. Di Battista, M. Patrignani, M. Pissonia, and M. Rimondini. Investigating prefix propagation through active BGP probing. In *IEEE ISCC*, 2006.

[13] R. Cohen and D. Raz. The Internet Dark Matter – on the Missing Links in the AS Connectivity Map. In *IEEE Infocom*, 2006.

[14] Internet routing registry, http://www.irr.net.

[15] K. Xu, Z. Duan, Z. Zhang, and J. Chandrashekar. On Properties of Internet Exchange Points and Their Impact on AS tolology and Relationship. In *Networking*, 2004.

[16] G. Siganos and M. Faloutsos. Analyzing BGP Policies: Methodology and Tool. In *IEEE Infocom*, 2004.

[17] Oregon routeview project, http://www.routeviews.org.

[18] Ripe route information service, http://www.ripe.net/ris.

[19] M. Crovella A. Lakhina, J. W. Byers and I. Matta. Sampling biases in ip topology measurements. In *IEEE Infocom*, 2003.

[20] D. Achlioptas, A. Clauset, D. Kempe, and C. Moore. On the bias of traceroute sampling, or power-law degree distributions in regular graphs. In *STOC*, 2005.

[21] Z. Mao, J. Rexford, J. Wang, and R. Katz. Towards an accurate AS-level traceroute tool. In *Sigcomm*, 2003.

[22] Z. Mao, D. Johnson, J. Rexford, J. Wang, and R. Katz. Scalable and accurate identification of AS-Level forwarding paths. In *Infocom*, 2004.

[23] Eran Shir. Personal communication via emails, Dec 2005.

[24] B. Donnet, P. Raoult, T. Friedman, and M. Crovella. Efficient algorithms for large-scale topology discovery. In *Proceedings of ACM SIGMETRICS*, June 2005.

[25] N. Spring, R. Mahajan, D. Wetherall, and T. Anderson. Measuring ISP topologies with rocketfuel. *IEEE/ACM Trans. Netw.*, 12(1):2–16, 2004.

[26] http://www.cs.ucr.edu/bgp.

[27] J. Xia and L. Gao. On the evaluation of as relationship inferences. In *IEEE Globecom*, November 2004.

[28] Skitter, http://www.caida.org/tools/measurement/skitter/.

[29] http://www.traceroute.org.

[30] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On Power-law Relationships of the Internet Topology. In *SIGCOMM*, 1999.

[31] A. Medina, I. Matta, and J. Byers. On the origin of powerlaws in Internet topologies. *CCR*, 30(2):18–34, April 2000.

[32] Q. Chen, H. Chang, R. Govindan, S. Jamin, S. Shenker, and W. Willinger. The Origin of Power Laws in Internet Topologies Revisited. In *Infocom*, 2002.

[33] S. Jaiswal, A. Rosenberg, and D. Towsley. Comparing the structure of power law graphs and the Internet AS graph. In *ICNP*, 2004.

[34] L. Gao and F. Wang. The extent of AS path inflation by routing policies. In *IEEE Global Internet*, 2000.

[35] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *ACM Sigcomm*, 2003.

[36] Package cleaning house, http://www.pch.net.

[37] L. Amimi, A. Shaikh, and H. Schulzrinne. Issues with inferring Internet topological attributes. In *SPIE*, July 2002.

[38] European internet exchange association, http://www.euro-ix.net.

The Flexlab Approach to Realistic Evaluation of Networked Systems

Robert Ricci   Jonathon Duerig   Pramod Sanaga   Daniel Gebhardt
Mike Hibler   Kevin Atkinson   Junxing Zhang   Sneha Kasera   Jay Lepreau

University of Utah, School of Computing

Abstract

Networked systems are often evaluated on overlay testbeds such as PlanetLab and emulation testbeds such as Emulab. Emulation testbeds give users great control over the host and network environments and offer easy reproducibility, but only artificial network conditions. Overlay testbeds provide real network conditions, but are not repeatable environments and provide less control over the experiment.

We describe the motivation, design, and implementation of Flexlab, a new testbed with the strengths of both overlay and emulation testbeds. It enhances an emulation testbed by providing the ability to integrate a wide variety of network models, including those obtained from an overlay network. We present three models that demonstrate its usefulness, including "application-centric Internet modeling" that we specifically developed for Flexlab. Its key idea is to run the application within the emulation testbed and use its offered load to measure the overlay network. These measurements are used to shape the emulated network. Results indicate that for evaluation of applications running over Internet paths, Flexlab with this model can yield far more realistic results than either PlanetLab without resource reservations, or Emulab without topological information.

1 Introduction

Public network testbeds have become staples of the networking and distributed systems research communities, and are widely used to evaluate prototypes of research systems in these fields. Today, these testbeds generally fall into two categories: *emulation testbeds* such as the emulation component of Emulab [37], which create artificial network conditions that match an experimenter's specification, and *overlay testbeds* such as PlanetLab [27], which send an experiment's traffic over the Internet. Each type of testbed has its own strengths and weaknesses. In this paper, we present Flexlab, which bridges the two types of testbeds, inheriting strengths from both.

Emulation testbeds such as Emulab and ModelNet [34] give users full control over the host and network environments of their experiments, enabling a wide range of experiments using different applications, network stacks, and operating systems. Experiments run on them are repeatable, to the extent that the application's behavior can be made deterministic. They are also well suited for developing and debugging applications—two activities that represent a large portion of the work in networked systems research and are especially challenging in the wide area [1, 31]. However, emulation testbeds have a serious shortcoming: their network conditions are artificial and thus do not exhibit some aspects of real production networks. Perhaps worse, researchers are *not sure* of two things: which network aspects are poorly modeled, and which of these aspects matter to their application. We believe these are two of the reasons researchers underuse emulation environments. That emulators are underused has also been observed by others [35].

Overlay testbeds, such as PlanetLab and the RON testbed [2], overcome this lack of network realism by sending experimental traffic over the real Internet. They can thus serve as a "trial by fire" for applications on today's Internet. They also have potential as a service platform for deployment to real end-users, a feature we do not attempt to replicate with Flexlab. However, these testbeds have their own drawbacks. First, they are typically overloaded, creating contention for host resources such as CPU, memory, and I/O bandwidth. This leads to a host environment that is unrepresentative of typical deployment scenarios. Second, while it may eventually be possible to isolate most of an experiment's host resources from other users of the testbed, it is impossible (by design) to isolate it from the Internet's varying conditions. This makes it fundamentally impossible to obtain repeatable results from an experiment. Finally, because hosts are shared among many users at once, users cannot perform many privileged operations including choosing the OS, controlling network stack parameters, and modifying the kernel.

Flexlab is a new testbed environment that combines the strengths of both overlay and emulation testbeds. In Flexlab, experimenters obtain networks that exhibit real Internet conditions *and* full, exclusive control over hosts. At the same time, Flexlab provides more control and repeatability than the Internet. We created this new environment by closely coupling an emulation testbed with an overlay testbed, using the overlay to provide network conditions for the emulator. Flexlab's modular framework qualitatively increases the range of network models that can be emulated. In this paper, we describe this framework and three models derived from the overlay testbed. These models are by no means the only models that can be built in the Flexlab framework, but they represent interesting points in the design space, and demonstrate the framework's flexibility. The first two use traditional network measurements in a straightforward fashion. The third, "application-centric Internet modeling" (ACIM), is a novel contribution itself.

ACIM stems directly from our desire to combine the strengths of emulation and live-Internet experimentation. We provide machines in an emulation testbed, and "import" network conditions from an overlay testbed. Our approach is application-centric in that it confines itself to the network conditions relevant to a particular application, using a simplified model of that application's own traffic to make its measurements on the overlay testbed. By doing this in near real-time, we create the illusion that network device interfaces in the emulator are distributed across the Internet.

Flexlab is built atop the most popular and advanced testbeds of each type, PlanetLab and Emulab, and exploits a public federated network data repository, the Datapository [3]. Flexlab is driven by Emulab testbed management software [36] that we recently enhanced to extend most of Emulab's experimentation tools to PlanetLab slivers, including automatic link tracing, distributed data collection, and control. Because Flexlab allows different network models to be "plugged in" without changing the experimenter's code or scripts, this testbed also makes it easy to compare and validate different network models.

This paper extends our previous workshop paper [9], and presents the following contributions:
(1) A software framework for incorporating a variety of highly-dynamic network models into Emulab;
(2) The ACIM emulation technique that provides high-fidelity emulation of live Internet paths;
(3) Techniques that infer available bandwidth from the TCP or UDP throughput of applications that do not continually saturate the network;
(4) An experimental evaluation of Flexlab and ACIM;
(5) A flexible network measurement system for PlanetLab. We demonstrate its use to drive emulations and construct simple models. We also present data that shows the significance on PlanetLab of non-stationary network conditions and shared bottlenecks, and of CPU scheduling delays.

Finally, Flexlab is currently deployed in Emulab in beta test, will soon be enabled for public production use, and will be part of an impending Emulab open source release.

2 Flexlab Architecture

The architecture of the Flexlab framework is shown in Figure 1. The application under test runs on emulator hosts, where the *application monitor* instruments its network operations. The application's traffic passes through the *path emulator*, which shapes it to introduce latency, limit bandwidth, and cause packet loss. The parameters for the path emulator are controlled by the *network model*, which may optionally take input from the monitor, from the *network measurement repository*, and from other sources. Flexlab's framework provides the ability to incorporate new network models, including highly dynamic ones, into Emulab. All parts of Flexlab except for the underlying emulation testbed are user-replaceable.

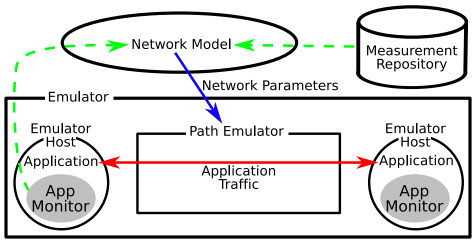


Figure 1: Architecture of the Flexlab framework. Any network model can be "plugged in," and can optionally use data from the application monitors or measurement repository.

2.1 Emulator

Flexlab runs on top of the Emulab testbed management system, which provides critical management infrastructure. It provides automated setup of emulated experiments by configuring hosts, switches, and path emulators within minutes. Emulab also provides a "full-service" interface for distributing experimental applications to nodes, controlling those applications, collecting packet traces, and gathering of log files and other results. These operations can be controlled and (optionally) fully automated by a flexible, secure event system. Emulab's portal extends all of these management benefits to PlanetLab nodes. This makes Emulab an ideal platform for Flexlab, as users can easily move back and forth between emulation, live experimentation, and Flexlab experimentation. New work [10] integrates a full experiment and data management system into Emulab—indeed, we used that "workbench" to gather and manage many of the results in this paper.

2.2 Application Monitor

The application monitor reports on the network operations performed by the application, such as the connections it makes, its packet sends and receives, and the socket options it sets. This information can be sent to the network model, which can use it to track which paths the application uses and discover the application's offered network load. Knowing the paths in use aids the network model by limiting the set of paths it must measure or compute; most applications will use only a small subset of the n^2 paths between n hosts. We describe the monitor in more detail later.

2.3 Path Emulator

The path emulator shapes traffic from the emulator hosts. It can, for example, queue packets to emulate delay, dequeue packets at a specific rate to control bandwidth, and drop packets from the end of the queue to emulate saturated router queues. Our path emulator is an enhanced version of FreeBSD's Dummynet [28]. We have made extensive improvements to Dummynet to add support for the features discussed in Section 5.2, as well as adding support for jitter and for several distributions: uniform, Poisson, and arbitrary distributions determined by user-supplied tables. Dummynet runs on separate hosts from the application, both to reduce contention for host resources, and so that applications can be run on any operating system.

For Flexlab we typically configure Dummynet so that it emulates a "cloud," abstracting the Internet as a set of per-flow pairwise network characteristics. This is a significant departure from Emulab's typical use: it is typically used with router-level topologies, although the topologies may be somewhat abstracted. The cloud model is necessary for us because our current models deal with end-to-end conditions, rather than trying to reverse engineer the Internet's router-level topology.

A second important piece of our path emulator is its control system. The path emulator can be controlled with Emulab's event system, which is built on a publish/subscribe system. "Delay agents" on the emulator nodes subscribe to events for the paths they are emulating, and update characteristics based on the events they receive. Any node can publish new characteristics for paths, which makes it easy to support both centralized and distributed implementations of network models. For example, control is equally easy by a single process that computes all model parameters or by a distributed system in which measurement agents independently compute the parameters for individual paths. The Emulab event system is lightweight, making it feasible to implement highly dynamic network models that send many events per second, and it is secure: event senders can affect only their own experiments.

2.4 Network Model

The network model supplies network conditions and parameters to the path emulator. The network model is the least-constrained component of the Flexlab architecture; the only constraint on a model implementation is that it must configure the path emulator through the event system. Thus, a wide variety of models can be created. A model may be static, setting network characteristics once at the beginning of an experiment, or dynamic, keeping them updated as the experiment proceeds. Dynamic network settings may be sent in real-time as the experiment proceeds, or the settings may be pre-computed and scheduled for delivery by Emulab's event scheduler.

We have implemented three distinct network models, discussed later. All of our models pair up each emulator node with a node in the overlay network, attempting to give the emulator node the same view of network characteristics as its peer in the overlay. The architecture, however, does not require that models come directly from overlay measurements. Flexlab can just as easily be used with network models from other sources, such as analytic models.

2.5 Measurement Repository

Flexlab's measurements are currently stored in Andersen and Feamster's Datapository. Information in the Datapository is available for use in constructing or parameterizing network models, and the networking community is encouraged to contribute their own measurements. We describe Flexlab's measurement system in the next section.

3 Wide-area Network Monitoring

Good measurements of Internet conditions are important in a testbed context for two reasons. First, they can be used as input for network models. Second, they can be used to select Internet paths that tend to exhibit a chosen set of properties. To collect such measurements, we developed and deployed a wide area network monitor, Flexmon. It has been running for a year, placing into the Datapository half a billion measurements of connectivity, latency, and bandwidth between PlanetLab hosts. Flexmon's design provides a measurement infrastructure that is shared, reliable, safe, adaptive, controllable, and accommodates high-performance data retrieval. Flexmon has some features in common with other measurement systems such as S^3 [39] and Scriptroute [32], but is designed for shared control over measurements and the specific integration needs of Flexlab.

Flexmon, shown in Figure 2, consists of five components: *path probers*, the *data collector*, the *manager*, *manager clients*, and the *auto-manager client*. A path prober runs on each PlanetLab node, receiving control commands from a central source, the manager. A command may change the measurement destination nodes, the type of measurement, and the frequency of measurement. Commands are sent by experimenters, using a manager client, or by the auto-manager client. The purpose of the auto-manager client is to maintain measurements between all PlanetLab sites. The auto-manager client chooses the least CPU-loaded node at each site to include in its measurement set, and makes needed changes as nodes and sites go up and down. The data collector runs on a server in Emulab, collecting measurement results from each path prober and storing them in the Datapository. To speed up both queries and updates, it contains a write-back cache in the form of a small database instance.

Due to the large number of paths between PlanetLab nodes, Flexmon measures each path at fairly low frequency—approximately every 2.5 hours for bandwidth, and 10 minutes for latency. To get more detail, experimenters can control Flexmon's measurement frequency of any path. Flexmon maintains a global picture of the net-

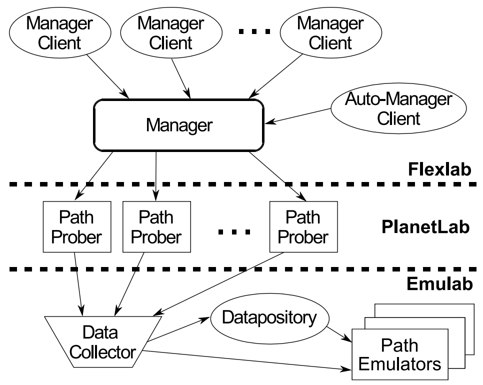


Figure 2: The components of Flexmon and their communication.

work resources it uses, and caps and adjusts the measurement rates to maintain safety to PlanetLab.

Flexmon currently uses simple tools to collect measurements: `iperf` for bandwidth, and `fping` for latency and connectivity. We had poor results from initial experiments with packet-pair and packet-train tools, including `pathload` and `pathchirp`. Our guiding principles thus far have been that the simpler the tool, the more reliable it typically is, and that the most accurate way of measuring the bandwidth available to a TCP stream is to use a TCP stream. Flexmon has been designed, however, so that it is relatively simple to plug in other measurement tools. For example, tools that trade accuracy for reduced network load or increased scalability [8, 13, 21, 23] could be used, or we could take opportunistic measurements of large file transfers by the CDNs on PlanetLab.

Flexmon's reliability is greatly improved by buffering results at each path prober until an acknowledgment is received from the data collector. Further speedup is possible by directly pushing new results to requesting Flexlab experiments instead of having them poll the database.

4 Simple Measurement-Driven Models

We have used measurements taken by Flexmon to build two simple, straightforward network models. These models represent incremental improvements over the way emulators are typically used today. Experimenters typically choose network parameters on an ad hoc basis and keep them constant throughout an experiment. Our *simple-static* model improves on this by using actual measured Internet conditions. The *simple-dynamic* model goes a step further by updating conditions as the experiment proceeds. Because the measurements used by these models are stored permanently in the Datapository, it is trivial to "replay" network conditions starting at any point in the past. Another benefit is that the simple models run entirely outside of the emulated environment itself, meaning that no restrictions are placed on the protocols, applications, or operating systems that run on the emulator hosts. The simple models do have some weaknesses, which we discuss in this section. These weaknesses are addressed by our more sophisticated model, ACIM, in Section 5.

4.1 Simple-static and Simple-dynamic

In both the simple-static and simple-dynamic models, each PlanetLab node in an experiment is associated with a corresponding emulation node in Emulab. A program called `dbmonitor` runs on an Emulab server, collecting path characteristics for each relevant Internet path from the Datapository. It applies the characteristics to the emulated network via the path emulator.

In simple-static mode, `dbmonitor` starts at the beginning of an experiment, reads the path characteristics from the DB, issues the appropriate events to the emulation agents, and exits. This model places minimal load on the path emulators and the emulated network, at the expense of fidelity. If the real path characteristics change during an experiment, the emulated network becomes inaccurate.

In simple-dynamic mode the experimenter controls the frequencies of measurement and emulator update. Before the experiment starts, `dbmonitor` commands Flexmon to increase the frequency of probing for the set of PlanetLab nodes involved in the experiment. Similarly, `dbmonitor` queries the DB and issues events to the emulator at the specified frequency, typically on the order of seconds. The dynamic model addresses some of the fidelity issues of the simple-static model, but it is still constrained by practical limits on measurement frequency.

4.2 Stationarity of Network Conditions

The simple models presented in this section are limited in the detail they can capture, due to a fundamental tension. We would like to take frequent measurements, to maximize the models' accuracy. However, if they are too frequent, measurements of overlapping paths (such as from a single source to several destinations) will necessarily overlap, causing interference that may perturb the network conditions. Thus, we must limit the measurement rate.

To estimate the effect that low measurement rates have on accuracy, we performed an experiment. We sent pings between pairs of nodes every 2 seconds for 30 minutes. We analyzed the latency distribution to find "change points" [33], which are times when the mean value of the latency samples changes. This statistical technique was used in a classic paper on Internet stationarity [41]; our method is similar to their "CP/Bootstrap" test. This analysis provides insight into the required measurement frequency—the more significant events missed, the poorer the accuracy of a measurement.

Table 1 shows some of the results from this test. We used representative nodes in Asia, Europe, and North America. One set of North American nodes was connected to the commercial Internet, and the other set to Internet2. The

Path	High	Low	Change
Asia to Asia	2	1	0.13%
Asia to Commercial	2	0	2.9%
Asia to Europe	4	0	0.5%
Asia to I2	6	0	0.59%
Commercial to Commercial	**20**	2	**39%**
Commercial to Europe	4	0	3.4%
Commercial to I2	**13**	1	**15%**
I2 to I2	4	0	0.02%
I2 to Europe	0	0	–
Europe to Europe	**9**	1	**12%**

Table 1: Change point analysis for latency.

first column shows the number of change points seen in this half hour. In the second column, we have simulated measurement at lower frequencies by sampling our high-rate data; we used only one of every ten measurements, yielding an effective sampling interval of 20 seconds. Finally, the third column shows the magnitude of the median change, in terms of the median latency for the path.

Several of the paths are largely stable with respect to latency, exhibiting few change points even with high-rate measurements, and the magnitude of the few changes is low. However, three of the paths (in bold) have a large number of change points, and those changes are of significant magnitude. In all cases, the low-frequency data misses almost all change points. In addition, we cannot be sure that our high-frequency measurements have found all change points. The lesson is that there are enough significant changes at small time scales to justify, and perhaps even necessitate, high-frequency measurements.

In Section 5, we describe application-centric Internet modeling, which addresses this accuracy problem by using the application's own traffic patterns to make measurements. In that case, the only load on the network, and the only self-interference induced, is that which would be caused by the application itself.

4.3 Modeling Shared Bottlenecks

There is a subtle complexity in network emulation based on path measurements of available bandwidth. This complexity arises when an application has multiple simultaneous network flows associated with a single node in the experiment. Because Flexmon obtains pairwise available bandwidth measurements using independent `iperf` runs, it does not reveal bottlenecks shared by multiple paths. Thus, independently modeling flows originating at the same host but terminating at different hosts can cause inaccuracies if there are shared bottlenecks. This is mitigated by the fact that if there is a high degree of statistical multiplexing on the shared bottleneck, interference by other flows dominates interference by the application's own flows [14]. In that case, modeling the application's flows as independent is still a reasonable approximation.

In the "cloud" configuration of Dummynet we model flows originating at the same host as being non-interfering.

	Sum of multiple TCP flows		
Path	1 flow	5 flows	10 flows
Commodity Internet Paths			
PCH to IRO	485 K	585 K	797 K
IRP to UCB-DSL	372 K	507 K	589 K
PBS to Arch. Tech.	348 K	909 K	952 K
Internet2 Paths			
Illinois to Columbia	3.95 M	9.05 M	9.46 M
Maryland to Calgary	3.09 M	15.4 M	30.4 M
Colorado St. to Ohio St.	225 K	1.20 M	1.96 M

Table 2: Available bandwidth estimated by multiple `iperf` flows, in bits per second. The PCH to IRO path is administratively limited to 10 megabits, and the IRP to UCB-DSL path is administratively limited to 1 megabit.

To understand how well this assumption holds, we measured multiple simultaneous flows on PlanetLab paths, shown in Table 2. For each path we ran three tests in sequence for 30 seconds each: a single TCP `iperf`, five TCP `iperfs` in parallel, and finally ten TCP `iperfs` in parallel. The reverse direction of each path, not shown, produced similar results.

Our experiment revealed a clear distinction between paths on the commodity Internet and those on Internet2 (I2). On the commodity Internet, running more TCP flows achieves only marginally higher aggregate throughput. On I2, however, five flows always achieve much higher throughput than one flow. In all but one case, ten flows also achieve significantly higher throughput than five. Thus, our previous assumption of non-interference between multiple flows holds true for the I2 paths tested, but not for the commodity Internet paths.

This difference may be a consequence of several possible factors. It could be due to the fundamental properties of these networks, including proximity of bottlenecks to the end hosts and differing degrees of statistical multiplexing. It could also be induced by peculiarities of PlanetLab. Some sites impose administrative limits on the amount of bandwidth PlanetLab hosts may use, PlanetLab attempts to enforce fair-share network usage between slices, and the TCP stack in the PlanetLab kernel is not tuned for high performance on links with high bandwidth-delay products (in particular, TCP window scaling is disabled).

To model this behavior, we developed additional simple Dummynet configurations. In the "shared" configuration, a node is assumed to have a single bottleneck that is shared by all of its outgoing paths, likely its last-mile link. In the "hybrid" configuration, some paths use the cloud model and others the shared model. The rules for hybrid are: If a node is an I2 node, it uses the cloud model for I2 destination nodes, and the shared model for all non-I2 destination nodes. Otherwise, it uses the shared model for all destinations. The bandwidth for shared pipes is set to the maximum found for any destination in the experiment. Flexlab users can select which Dummynet configuration to use.

Clearly, more sophisticated shared-bottleneck models are possible for the simple models. For example, it might be possible to identify bottlenecks with Internet tomography, such as iPlane [21]. Our ACIM model, discussed next, takes a completely different approach to the shared-bottleneck problem.

5 Application-Centric Internet Modeling

The limitations of our simple models lead us to develop a more complex technique, *application-centric Internet modeling*. The difficulties in simulating or emulating the Internet are well known [12, 20], though progress is continually made. Likewise, creating good *general-purpose* models of the Internet is still an open problem [11]. While progress has been made on measuring and modeling aspects of the Internet sufficient for certain uses, such as improving overlay routing or particular applications [21, 22], the key difficulty we face is that a general-purpose emulator, in theory, has a stringent accuracy criterion: it must yield accurate results for *any* measurement of *any* workload.

ACIM approaches the problem by *modeling the Internet as perceived by the application*—as viewed through its limited lens. We do this by running the application and Internet measurements simultaneously, using the application's behavior *running inside Emulab* to generate traffic *on PlanetLab* and collect network measurements. The network conditions experienced by this replicated traffic are then applied, in near real-time, to the application's emulated network environment.

ACIM has five primary benefits. The first is in terms of node and path scaling. A particular instance of any application will use a tiny fraction of all of the Internet's paths. By confining measurement and modeling only to those paths that the application actually uses, the task becomes more tractable. Second, we avoid numerous measurement and modeling problems, by assessing end-to-end behavior rather than trying to model the intricacies of the network core. For example, we do not need precise information on routes and types of outages—we need only measure their effects, such as packet loss and high latency, on the application. Third, rare or transient network effects are immediately visible to the application. Fourth, it yields accurate information on how the network will react to the offered load, automatically taking into account factors that are difficult or impossible to measure without direct access to the bottleneck router. These factors include the degree of statistical multiplexing, differences in TCP implementations and RTTs of the cross traffic, the router's queuing discipline, and unresponsive flows. Fifth, it tracks conditions quickly, by creating a feedback loop which contiually adjusts offered loads and emulator settings in near real-time.

ACIM is *precise* because it assesses only relevent parts of the network, and it is *complete* because it automatically

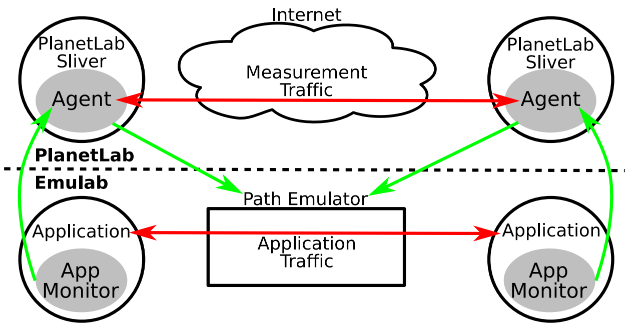


Figure 3: The architecture and data flow of application-centric Internet modeling.

accounts for all potential network-related behavior. (Of course, it is precise in terms of paths, not traffic.) Its concrete approach to modeling and its level of fidelity should provide an environment that experimenters can trust when they do not know their application's dependencies.

Our technique makes two common assumptions about the Internet: that the location of the bottleneck link does not change rapidly (though its characteristics may), and that most packet loss is caused by congestion, either due to cross traffic or its own traffic. In the next section, we first concentrate on TCP flows, then explain how we have extended the concepts to UDP.

5.1 Architecture

We pair each node in the emulated network with a peer in the live network, as shown in Figure 3. The portion of this figure that runs on PlanetLab fits into the "network model" portion of the Flexlab architecture shown in Figure 1. The ACIM architecture consists of three basic parts: an application monitor which runs on Emulab nodes, a measurement agent which runs on PlanetLab nodes, and a path emulator connecting the Emulab nodes. The agent receives characteristics of the application's offered load from the monitor, replicates that load on PlanetLab, determines path characteristics through analysis of the resulting TCP stream, and sends the results back into the path emulator as traffic shaping parameters. We now detail each of these parts.

Application Monitor on Emulab. The application monitor runs on each node in the emulator and tracks the network calls made by the application under test. It tracks the application's network activity, such as connections made and data sent on those connections. The monitor uses this information to create a simple model of the offered network load and sends this model to the measurement agent on the corresponding PlanetLab node. The monitor supports both TCP and UDP sockets. It also reports on important socket options, such as socket buffer sizes and the state of TCP's `TCP_NODELAY` flag.

We instrument the application under test by linking it with a library we created called `libnetmon`. This library's purpose is to provide the model with information about the application's network behavior. It wraps network system calls such as `connect()`, `accept()`, `send()`, `sendto()`, and `setsockopt()`, and informs the application monitor of these calls. In many cases, it summarizes: for example, we do not track the full contents of `send()` calls, simply their sizes and times. `libnetmon` can be dynamically linked into a program using the `LD_PRELOAD` feature of modern operating systems, meaning that most applications can be run without modification. We have tested `libnetmon` with a variety of applications, ranging from `iperf` to Mozilla Firefox to Sun's JVM.

By instrumenting the application directly, rather than snooping on network packets it puts on the wire, we are able to measure the application's *offered load* rather than simply the *throughput achieved*. This distinction is important, because the throughput achieved is, at least in part, a function of the parameters the model has given to the path emulator. Thus, we cannot assume that what an application is *able* to do is the same as what it is *attempting* to do. If, for example, the available bandwidth on an Internet path increases, so that it becomes greater than the bandwidth setting of the corresponding path emulator, offering only the achieved throughput on this path would fail to find the additional available bandwidth.

Measurement Agent on PlanetLab. The measurement agent runs on PlanetLab nodes, and receives information from the application monitor about the application's network operations. Whenever the application running on Emulab connects to one of its peers (also running inside Emulab), the measurement agent likewise connects to the agent representing the peer. The agent uses the simple model obtained by the monitor to generate similar network load; the monitor keeps the agent informed of the `send()` and `sendto()` calls made by the application, including the amount of data written and the time between calls. The agent uses this information to recreate the application's network behavior, by making analogous `send()` calls. Note that the offered load model does not include the application's packet payload, making it relatively lightweight to send from the monitor to the agent.

The agent uses `libpcap` to inspect the resulting packet stream and derive network conditions. For every ACK it receives from the remote agent, it calculates instantaneous throughput and RTT. For TCP, we use TCP's own ACKs, and for UDP, we add our own application-layer ACKs. The agent uses these measurements to generate parameters for the path emulator, discussed below.

5.2 Inference and Emulation of Path Conditions

Our path emulator is an enhanced version of the Dummynet traffic shaper. We emulate the behavior of the bottleneck router's queue within this shaper as shown in Figure 4. Dummynet uses two queues: a bandwidth queue, which emulates queuing delay, and a delay queue, which models

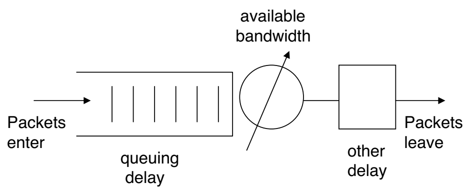


Figure 4: Path emulation

all other sources of delay, such as propagation, processing, and transmission delays. Thus, there are three important parameters: the size of the bandwidth queue, the rate at which it drains, and the length of time spent in the delay queue. Since we assume that most packet loss is caused by congestion, we induce loss only by limiting the size of the bandwidth queue and the rate it drains.

Because the techniques in this section require that there be application traffic to measure, we use the simple-static model to set initial conditions for each path. They will only be experienced by the first few packets; after that, ACIM provides higher-quality measurements.

Bandwidth Queue Size. The bandwidth queue has a finite size, and when it is full, packets arriving at the queue are dropped. The bottleneck router has a queue whose maximum capacity is measured in terms of bytes and/or packets, but it is difficult to directly measure either of these capacities. Sommers et al. [29] proposed using the maximum one-way delay as an approximation of the size of the bottleneck queue. This approach is problematic on PlanetLab because of the difficulty of synchronizing clocks, which is required to calculate one-way delay. Instead, we approximate the size of the queue in terms of *time*—that is, the longest time one of our packets has spent in the queue without being dropped. We assume that congestion will happen mostly along the forward edge of a network path, and thus can approximate the maximum queuing delay by subtracting the minimum RTT from the maximum RTT. We refine this number by finding the maximum queuing delay just before a loss event.

Available Bandwidth. TCP's fairness (the fraction of the capacity each flow receives) is affected by differences in the RTTs of flows sharing the link [18]. Measuring the RTTs of flows we cannot directly observe is difficult or impossible. Thus, the most accurate way to determine how the network will react to the load offered by a new flow is to offer that load and observe the resulting path properties.

We observe the inter-send times of acknowledgment packets and the number of bytes acknowledged by each packet to determine the instantaneous goodput of a connection: $goodput = (bytes\ acked)/(time\ since\ last\ ack)$. We then estimate the throughput of a TCP connection between PlanetLab nodes by computing a moving average of the instantaneous goodput measurements for the preceding half-second. This averages out any outliers, allowing for a more consistent metric.

This measurement takes into account the reactivity of other flows in the network. While calculating this goodput is straightforward, there are subtleties in mapping to available bandwidth. The traffic generated by the measurement agent may not fully utilize the available bandwidth. For instance, if the load generated by the application is lower than the available bandwidth, or TCP fills the receive window, the throughput does not represent available bandwidth. When this situation is detected, we should not cap the emulator bandwidth to that artificially slow rate. Thus, we *lower* the bandwidth used by the emulator only if we detect that we are fully loading the PlanetLab path. If we see a goodput that is higher than the goodput when we last saturated the link, then the available bandwidth must have increased, and we *raise* the emulator bandwidth.

Queuing theory shows that when a buffered link is overutilized, the time each packet spends in the queue, and thus the observed RTT, increases for each successive packet. Additionally, `send()` calls tend to block when the application is sending at a rate sufficient to saturate the bottleneck link. In practice, since each of these signals is noisy, we use a combination of them to determine when the bottleneck link is saturated. To determine whether RTT is increasing or decreasing, we find the slope of RTT vs. sample number using least squares linear regression.

Other Delay. The measurement agent takes fine-grained latency measurements. It records the time each packet is sent, and when it receives an ACK for that packet, calculates the RTT seen by the most recent acknowledged packet. For the purposes of emulation, we calculate the "Base RTT" the same way as TCP Vegas [5]: that is, the minimum RTT recently seen. This minimum delay accounts for the propagation, processing, and transmission delays along the path with a minimum of influence by queuing delay.

We set the delay queue's delay to half the base RTT to avoid double-counting queuing latency, which is modeled in the bandwidth queue.

Outages and Rare Events. There are many sources of outages and other anomalies in network characteristics. These include routing anomalies, link failures, and router failures. Work such as PlanetSeer [40] and numerous BGP studies seeks to explain the causes of these anomalies. Our application-centric model has an easier task: to faithfully reproduce the effect of these rare events, rather than finding the underlying cause. Thus, we observe the features of these rare events that are *relevant* to the application. Outages can affect Flexlab's control plane, however, by cutting off Emulab from one or more PlanetLab nodes. In future work, we can improve robustness by using an overlay network such as RON [2].

Per-Flow Emulation. In our application-centric model, the path emulator is used to shape traffic on a per-flow rather than a per-path basis. If there is more than one flow using a path, the bandwidth seen by each flow depends on many variables, including the degree of statistical multiplexing on the bottleneck link, when the flows begin, and the queuing policy on the bottleneck router. We let this contention for resources occur in the overlay network, and reflect the results into the emulator by per-flow shaping.

5.3 UDP Sockets

ACIM for UDP differs in some respects from ACIM for TCP. The chief difference is that there are no protocol-level ACKs in UDP. We have implemented a custom application-layer protocol on top of UDP that adds the ACKs needed for measuring RTT and throughput. This change affects only the replication and measurement of UDP flows; path emulation remains unchanged.

Application Layer Protocol. Whereas the TCP ACIM sends random payloads in its measurement packets, UDP ACIM runs an application-layer protocol on top of them. The protocol embeds sequence numbers in the packets on the forward path, and on the reverse path, sequence numbers and timestamps acknowledge received packets. Our protocol requires packets to be at least 57 bytes long; if the application sends packets smaller than this, the measurement traffic uses 57-byte packets.

Unlike TCP, our UDP acknowledgements are selective, not cumulative, and we also do not retransmit lost packets. We do not need *all* measurement traffic to get through, we simply measure how much does. An ACK packet is sent for every data packet received, but each ACK packet contains ACKs for several recent data packets. This redundancy allows us to get accurate bandwidth numbers without re-sending lost packets, and works in the face of moderate ACK packet loss.

Available Bandwidth. Whenever an ACK packet is received at the sender, goodput is calculated as $g = s/(t_n - t_{n-1})$, where g is goodput, s is the size of the data being acknowledged, t_n is the receiver timestamp for the current ACK, and t_{n-1} is the last receiver ACK timestamp received. By using inter-packet timings from the receiver, we avoid including jitter on the ACK path in our calculations, and the clocks at the sender and receiver need not be synchronized. Throughput is calculated as a moving average over the last 100 acknowledged packets or half second, whichever is less. If any packet loss has been detected, this throughput value is fed to the application monitor as the available bandwidth on the forward path.

Delay measurements. Base RTT and queuing delay are computed the same way for UDP as they are for TCP.

Reordering and Packet Loss. Because TCP acknowledgements are cumulative, reordering of packets on the forward path is implicitly taken care of. We have to handle it explicitly in the case of UDP. Our UDP measurement protocol can detect packet reordering in both directions. Because each ACK packet carries redundant ACKs, reordering on the reverse path is not of concern. A data packet is considered to be lost if ten packets sent after it have been acknowledge. It is also considered lost if the difference between the receipt time of the latest ACK and the send time of the data packet is greater than $10 \cdot (average\ RTT + 4 \cdot standard\ deviation\ of\ recent\ RTTs)$.

5.4 Challenges

Although the design of ACIM is straightforward when viewed at a high level, there are a host of complications that limit the accuracy of the system. Each was a significant barrier to implementation; we describe two.

Libpcap Loss. We monitor the connections on the measurement agent with `libpcap`. The `libpcap` library copies a part of each packet as it arrives or leaves the (virtual) interface and stores them in a buffer pending a query by the application. If packets are added to this buffer faster than they are removed by the application, some of them may be dropped. The scheduling behavior described in Appendix A is a common cause of this occurrence, as processes can be starved of CPU for hundreds of milliseconds. These dropped packets are still seen by the TCP stack in the kernel, but they are not seen by the application.

This poses two problems. First, we found it not uncommon for all packets over a long period of time (up to a second) to be dropped by the `libpcap` buffer. In this case it is impossible to know what has occurred during that period. The connection may have been fully utilizing its available bandwidth or it may have been idle during part of that time, and there is no way to reliably tell the difference. Second, if only one or a few packets are dropped by the `libpcap` buffer, the "falseness" of the drops may not be detectable and may skew the calculations.

Our approach is to reset our measurements after periods of detected loss, no matter how small. This avoids the potential hazards of averaging measurements over a period of time when the activity of the connection is unknown. The downside is that in such a situation, a change in bandwidth would not be detected as quickly and we may average measurements over non-contiguous periods of time. We know of no way to reliably detect which stream(s) a `libpcap` loss has affected in all cases, so we must accept that there are inevitable limits to our accuracy.

Ack Bursts. Some paths on PlanetLab have anomalous behaviors. The most severe example of this is a path that delivers bursts of acknowledgments over small timescales. In one case, acks that were sent over a period of 12 milliseconds arrived over a period of less than a millisecond, an order of magnitude difference. This caused some over-estimation of delay (by up to 20%), and an order of magnitude over-estimation of throughput. We cope with this phenomenon in two ways. First, we use TCP timestamps to obtain the ACK inter-departure times on the receiver rather than the ACK inter-arrival times on the sender. This technique corrects for congestion and other anomalies on the reverse path. Second, we lengthened the period over which we average (to about 0.5 seconds), which is also needed to dampen excessive jitter.

6 Evaluation

We evaluate Flexlab by presenting experimental results from three microbenchmarks and a real application. Our results show that Flexlab is more faithful than simple emulation, and can remove artifacts of PlanetLab host conditions. Doing a rigorous *validation* of Flexlab is extremely difficult, because it seems impossible to establish ground truth: each environment being compared can introduce its own artifacts. Shared PlanetLab nodes can hurt performance, experiments on the live Internet are fundamentally unrepeatable, and Flexlab might introduce artifacts through its measurement or path emulation. With this caveat, our results show that for at least some complex applications running over the Internet, Flexlab with ACIM produces more accurate and realistic results than running with the host resources typically available on PlanetLab, or in Emulab without network topology information.

6.1 Microbenchmarks

We evaluate ACIM's detailed fidelity using `iperf`, a standard measurement tool that simulates bulk data transfers. `iperf`'s simplicity makes it ideal for microbenchmarks, as its behavior is consistent between runs. With TCP, it simply sends data at the fastest possible rate, while with UDP it sends at a specified constant rate. The TCP version is, of course, highly reactive to network changes.

As in all of our experiments, each application tested on PlanetLab and each major Flexlab component (measurement agent, Flexmon) are run in separate slices.

6.1.1 TCP iperf and Cross-Traffic

Figure 5 shows the throughput of a representative two minute run in Flexlab of `iperf` using TCP. The top graph shows throughput achieved by the measurement agent, which replicated `iperf`'s offered load on the Internet between AT&T and the Univ. of Texas at Arlington. The bottom graph shows the throughput of `iperf` itself, running on an emulated path and dedicated hosts inside Flexlab.

To induce a change in available bandwidth, between times 35 and 95 we sent cross-traffic on the Internet path, in the form of ten `iperf` streams between other PlanetLab nodes at the same sites. Flexlab closely tracks the changed bandwidth, bringing the throughput of the path emulator down to the new level of available bandwidth. It also tracks network changes that we did not induce, such as the one at time 23. However, brief but large drops in throughput occasionally occur in the PlanetLab graph but not the Flexlab

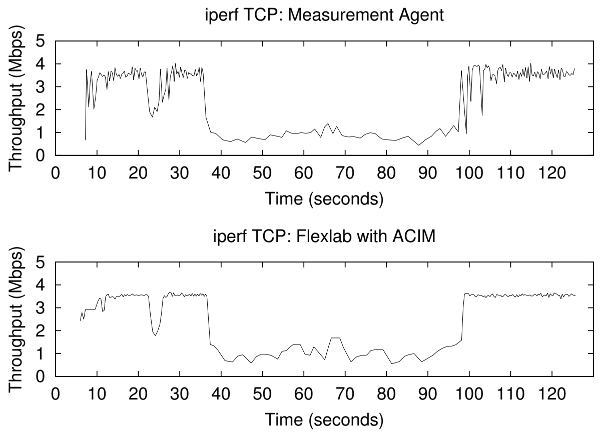


Figure 5: Application-centric Internet modeling, comparing agent throughput on PlanetLab (top) with the throughput of the application running in Emulab and interacting with the model (bottom).

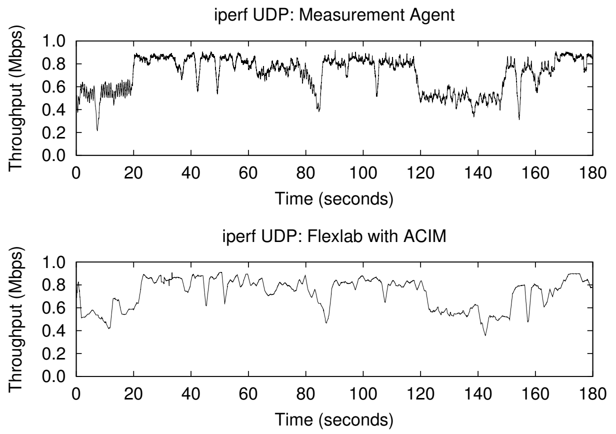


Figure 7: The UDP throughput of `iperf` (below) compared with the actual throughput successfully sent by the measurement agent (above) when using the ACIM model in Flexlab.

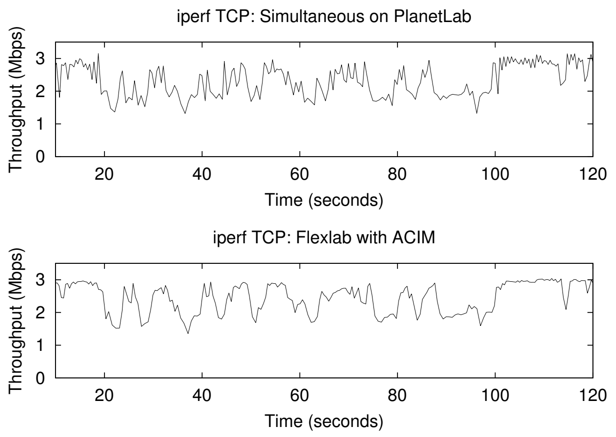


Figure 6: Comparison of the throughput of a TCP `iperf` running on PlanetLab (top) with a TCP `iperf` simultaneously running under Flexlab with ACIM (bottom).

graph, such as those starting at time 100. Through log file analysis we determined that these drops are due to temporary CPU starvation on PlanetLab, preventing even the lightweight measurement agent from sustaining the sending rate of the real application. These throughput drops demonstrate the impact of the PlanetLab scheduling delays documented in Appendix A. The agent correctly determines that these reductions in throughput are not due to available bandwidth changes, and deliberately avoids mirroring these PlanetLab host artifacts on the emulated path. Finally, the measurement agent's throughput exhibits more jitter than the application's, showing that we could probably further improve ACIM by adding a jitter model.

6.1.2 Simultaneous TCP iperf Runs

ACIM is designed to subject an application in the emulator to the same network conditions that application would see on the Internet. To evaluate how well ACIM meets this goal, we compared two instances of `iperf`: one on PlanetLab, and one in Flexlab. Because we cannot expect runs done on the Internet at different times to show the same results, we ran these two instances simultanously. The top graph in Figure 6 shows the throughput of `iperf` run directly on PlanetLab between NEC Labs and Intel Research Seattle. The bottom graph shows the throughput of another `iperf` run at the same time in Flexlab, between the same "hosts." As network characteristics vary over the connection's lifetime, the throughput graphs correspond impressively. The average throughputs are close: PlanetLab was 2.30 Mbps, while Flexlab was 2.41 Mbps (4.8% higher). These results strongly suggest that ACIM has high fidelity. The small difference may be due to CPU load on PlanetLab; we speculate that difference is small because `iperf` consumes few host resources, unlike a real application on which we report shortly.

6.1.3 UDP iperf

We have made an initial evaluation of the UDP ACIM support, which is newer than our TCP support. We used a single `iperf` to generate a 900 Kbps UDP stream. As in Sec. 6.1.1, we measured the throughput achieved by both the measurement agent on PlanetLab and the `iperf` stream running on Flexlab. The graphs in Figure 7 closely track each other. The mean throughputs are close: 746 Kbps for Iperf, and 736 Kbps for the measurement agent, 1.3% lower. We made three similar runs between these nodes, at target rates varying from 800–1200 Kbps. The differences in mean throughput were similar: -2.5%, 0.4%, and 4.4%. ACIM's UDP accuracy appears very good in this range. A more thorough evaluation is future work.

6.2 Macrobenchmark: BitTorrent

This next set of experiments demonstrates several things: first, that Flexlab is able to handle a real, complex, distributed system that is of interest to researchers; second, that PlanetLab host conditions can make an enormous impact on the network performance of real applications; third, that both Flexlab and PlanetLab with host CPU reservations give similar and likely accurate results; and fourth, preliminary results indicate that our simple static models of the Internet don't (yet) provide high-fidelity emulation.

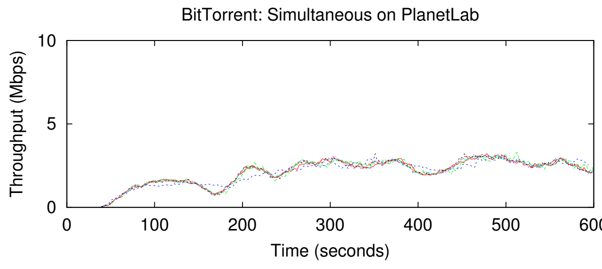
BitTorrent (BT) is a popular peer-to-peer program for cooperatively downloading large files. Peers act as both clients and servers: once a peer has downloaded part of a file, it serves that to other peers. We modified BT to use a static tracker to remove some–but by no means all—sources of non-determinism from repeated BT runs. Each experiment consisted of a seeder and seven BT clients, each located at a different site on Internet2 or GÉANT, the European research network.[1] We ran the experiments for 600 seconds, using a file that was large enough that no client could finish downloading it in that period.

6.2.1 ACIM vs. PlanetLab

We began by running BT in a manner similar to the simultaneous `iperf` microbenchmark described in Sec. 6.1.2. We ran two instances of BT simultaneously: one on PlanetLab and one using ACIM on Flexlab. These two sets of clients did not communicate directly, but they did compete for bandwidth on the same paths: the PlanetLab BT directly sends traffic on the paths, while the Flexlab BT causes the measurement agent to send traffic on those same paths.

Figure 8 shows the download rates of the BT clients, with the PlanetLab clients in the top graph, and the Flexlab clients in the bottom. Each line represents the download rate of a single client, averaged over a moving window of 30 seconds. The PlanetLab clients were only able to sustain an average download rate of 2.08 Mbps, whereas those on Flexlab averaged triple that rate, 6.33 Mbps. The download rates of the PlanetLab clients also clustered much more tightly than in Flexlab. A series of runs showed that the clustering was consistent behavior. Table 3 summarizes those runs, and shows that the throughput differences were also repeatable, but with Flexlab higher by a factor of 2.5 instead of 3.

[1]The sites were stanford.edu (10Mb), uoregon.edu (10Mb), cmu.edu (5Mb), usf.edu, utep.edu, kscy.internet2.planet-lab.org, uni-klu.ac.at, and tssg.org. The last two are on GÉANT; the rest on I2. Only the first three had imposed bandwidth limits. All ran PlanetLab 3.3, which contained a bug which enforced the BW limits even between I2 sites. We used the official BT program v. 4.4.0, which is in Python. All BT runs occurred in February 2007. 5 & 15 minute load averages for all nodes except the seeder were typically 1.5 (range 0.5–5); the seed (Stanford) had a loadavg of 14–29, but runs with a less loaded seeder gave similar results. Flexlab/Emulab hosts were all "pc3000"s: 3.0 Ghz Xeon, 2GB RAM, 10K RPM SCSI disk.



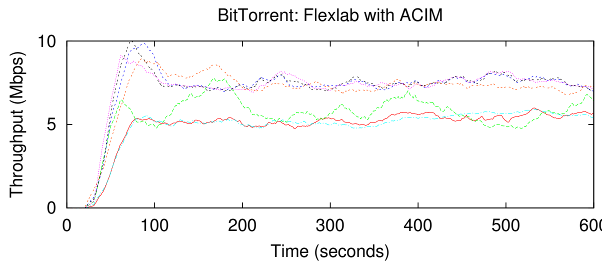
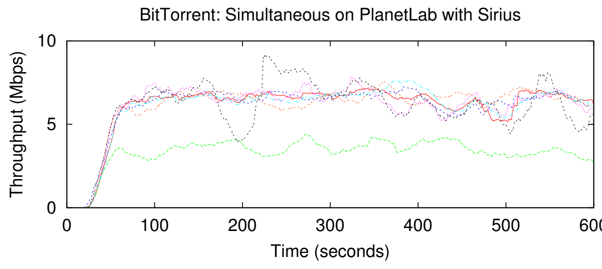


Figure 8: A comparison of download rates of BT running simultaneously on PlanetLab (top) and Flexlab using ACIM (bottom). The seven clients in the PlanetLab graph are tightly clustered.



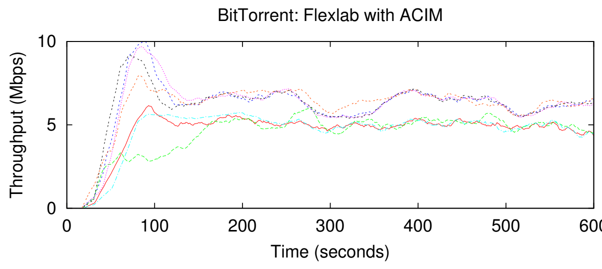


Figure 9: Download rates of BT simultaneously running on PlanetLab with Sirius (top), compared to Flexlab ACIM (bottom).

These results, combined with the accuracy of the microbenchmarks, suggest that BT's throughput on PlanetLab is constrained by host overload not found in Flexlab. Our next experiment attempts to test this hypothesis.

6.2.2 ACIM vs. PlanetLab with Sirius

Sirius is a CPU and bandwidth reservation system for PlanetLab. It ensures that a sliver receives at least 25% of its host's CPU, but does not give priority access to other host resources such as disk I/O or RAM. Normal Sirius also includes a bandwidth reservation feature, but to isolate the ef-

Experiment	Flexlab	PlanetLab	Ratio
No Sirius (6 runs)	5.78 (0.072)	2.27 (0.074)	2.55 (0.088)
Sirius (5 runs)	5.44 (0.29)	5.24 (0.34)	1.04 (0.045)

Table 3: Mean BT download rate in Mbps and std. dev. (in parentheses) of multiple Flexlab and PlanetLab runs, as in Sec. 6.2. Since these were run at a different time, network conditions may have changed.

fects of CPU sharing, we had PlanetLab operations disable this feature in our Sirius slice. Currently, only one slice, PlanetLab-wide, can have a Sirius reservation at a time. By using Sirius, we reduce the potential for PlanetLab host artifacts and get a better sense of Flexlab's accuracy.

We repeated the previous experiment fifteen minutes later, with the sole difference that the PlanetLab BT used Sirius. We ran BT on Flexlab at the same time; its measurement agent on PlanetLab did not have the benefit of Sirius. Figure 9 shows the download rates of these simultaneous runs. Sirius more than doubled the PlanetLab download rate of our previous PlanetLab experiment, from 2.08 to 5.80 Mbps. This demonstrates that BT is highly sensitive to CPU availability, and that the CPU typically available on PlanetLab is insufficient to produce accurate results for some complex applications. It also highlights the need for sufficient, reserved host resources on current and future network testbeds. In this run, the Flexlab and PlanetLab download rates are within 4% of each other, at 5.56 Mbps and 5.80 Mbps, respectively. These results are consistent, as shown by repeated experiments in Table 3. This indicates that Flexlab with ACIM provides a good environment for running experiments that need PlanetLab-like network conditions without host artifacts.

Resource Use. To estimate the host resources consumed by BT and the measurement agent we ran Flexlab with a "fake PlanetLab" side that ran inside Emulab. The agent took only 2.6% of the CPU, while BT took 37–76%, a factor of 14–28 higher. The agent's resident memory use was about 2.0MB, while BT used 8.4MB, a factor of 4 greater.

6.2.3 Simple Static Model

We ran BT again, this time using the simple-static model outlined in Sec. 4.1. Network conditions were those collected by Flexmon five minutes before running the BT experiment in Sec. 6.2.1, so we would hope to see a mean download rate similar to ACIM's: 6.3 Mbps.[2] We did three runs using the "cloud," "shared," and "hybrid" Dummynet configurations. We were surprised to find that the shared configuration gave the best approximation of BT's behavior on PlanetLab. The cloud configuration resulted in very high download rates (12.5 Mbps average), and the rates showed virtually no variation over time. Because six of the eight nodes used for our BT experiments are on I2, the hybrid configuration made little difference. The two GÉANT nodes now had realistic (lower) download rates, but the overall mean was still 10.7 Mbps. The shared configuration produced download rates that varied on timescales similar to those we have seen on PlanetLab and with ACIM. While the mean download rate was more accurate than the other configurations, it was 25% lower than that we would expect, at 5.1 Mbps.

This shows that the shared bottleneck models we developed for the simple models are not yet sophisticated enough to provide high fidelity emulation. The cloud configuration seems to under-estimate the effects of shared bottlenecks, and the shared configuration seems to over-estimate them, though to a lesser degree. Much more study is needed of these models and PlanetLab's network characteristics.

7 Related Work

Network measurement to understand and model network behavior is a popular research area. There is an enormous amount of related work on measuring and modeling Internet characteristics including bottleneck-link capacity, available bandwidth, packet delay and loss, topology, and more recently, network anomalies. Examples include [7, 8, 30, 17, 29, 38]. In addition to their use for evaluating protocols and applications, network measurements and models are used for maintaining overlays [2] and even for offering an "underlay" service [22]. PlanetLab has attracted many measurement studies specific to it [31, 19, 40, 25]. Earlier, Zhang et al. [41] showed that there is significant stationarity of Internet path properties, but argued that this alone does not mean that the latency characteristics important to a particular application can be sufficiently modeled with a stationary model.

Monkey [6] collects live TCP traces near servers, to faithfully replay client workload. It infers some network characteristics. However, Monkey is tied to a web server environment, and does not easily generalize to arbitrary TCP applications. Jaisal et al. did passive inference of TCP connection characteristics [15], but focused on other goals, including distinguishing between TCP implementations.

Trace-Based Mobile Network Emulation [24] has similarities to our work, in that it used traces from mobile wireless devices to develop models to control a synthetic networking environment. However, it emphasizes production of a parameterized model, and was intended to collect application-independent data for specific paths taken by mobile wireless nodes. In contrast, we concentrate on measuring ongoing Internet conditions, and our key model

[2]The 6.2.1 experiment differed from this one in that the former generated traffic on PlanetLab from two simultaneous BT's, while this experiment ran only one BT at a time. This unfortunate methodological difference could explain much of the difference between ACIM and the simple cloud model, but only if the simultaneous BT's in 6.2.1 significantly affected each other. That seemed unlikely due to the high degree of stat muxing we expect on I2 (and probably GÉANT) paths, both apriori and from the results in Sec. 4.3. However, that assumption needs study.

is application-centric.

Overlay Networks. Our ACIM approach can be viewed as a highly unusual sort of overlay network. In contrast to typical overlays designed to provide resilient or optimized services, our goal is to provide realism—to *expose* rather than mitigate the effects of the Internet. A significant practical goal of our project is to provide an experimentation platform for the development and evaluation of "traditional" overlay networks and services. By providing an environment that emulates real-world conditions, we enable the study of new overlay technologies designed to deal with the challenges of production networks.

Although our aims differ from those of typical overlay networks, we share a common need for measurement. Recent projects have explored the provision of common measurement and other services to support overlay networks [21, 22, 16, 26]. These are exactly the types of models and measurement services that our new testbed is designed to accept.

Finally, both VINI [4] and Flexlab claim "realism" and "control" as primary goals, but their kinds of realism and control are almost entirely different. The realism in VINI is that it peers with real ISPs so it can potentially carry real end-user traffic. The control in VINI is experimenter-controlled routing, forwarding, and fault injection, and provision of some dedicated links. In contrast, the realism in Flexlab is real, variable Internet conditions and dedicated hosts. The control in Flexlab is over pluggable network models, the entire hardware and software of the hosts, and rich experiment control.

8 Conclusion

Flexlab is a new experimental environment that provides a flexible combination of network model, realism, and control, and offers the potential for a friendly development and debugging environment. Significant work remains before Flexlab is a truly friendly environment, since it has to cope with the vagaries of a wide-area and overloaded system, PlanetLab. Challenging work also remains to extensively validate and likely refine application-centric Internet modeling, especially UDP.

Our results show that an end-to-end model, ACIM, achieves high fidelity. In contrast, simple models that exploit only a small amount of topology information (commodity Internet vs. Internet2) seem insufficient to produce an accurate emulation. That presents an opportunity to apply current and future network tomography techniques. When combined with data, models, and tools from the vibrant measurement and modeling community, we believe Flexlab with new models, not just ACIM, will be of great use to researchers in networking and distributed systems.

Acknowledgments: We are grateful to our co-workers for much implementation, evaluation, operations, discussion, design, and some writing: Sachin Goyal, David Johnson, Tim Stack, Kirk Webb, Eric Eide, Vaibhave Agarwal, Russ Fish, Leigh Stoller, and Venkat Chakravarthy; to our shepherd Srini Seshan, the reviewers, and Ken Yocum for their many useful comments, to Dave Andersen and Nick Feamster for the Datapository, to Dave for helpful discussion, to David Eisenstat and the PlanetLab team for their help and cooperation, to Vivek Pai and KyoungSoo Park for offering access to CoDeen measurements, to Jane-Ellen Long for her patience, and to NSF for its support under grants CNS–0335296, CNS–0205702, and CNS–0338785.

References

[1] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab Application Management Using Plush. *ACM SIGOPS OSR*, 40(1):33–40, Jan. 2006.

[2] D. Andersen et al. Resilient Overlay Networks. In *Proc. SOSP*, pages 131–145, Mar. 2001.

[3] D. G. Andersen and N. Feamster. Challenges and Opportunities in Internet Data Mining. Technical Report CMU–PDL–06–102, CMU Parallel Data Laboratory, Jan. 2006.

[4] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *Proc. SIGCOMM*, pages 3–14, Sept. 2006.

[5] L. Brakmo, S. O'Malley, and L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM*, pages 24–35, Aug.–Sept. 1994.

[6] Y.-C. Cheng et al. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proc. USENIX*, pages 87–98, June–July 2004.

[7] M. Coates, A. O. Hero III, R. Nowak, and B. Yu. Internet Tomography. *IEEE Signal Processing Mag.*, 19(3):47–65, May 2002.

[8] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *Proc. SIGCOMM*, pages 15–26, Aug.–Sept. 2004.

[9] J. Duerig, R. Ricci, J. Zhang, D. Gebhardt, S. Kasera, and J. Lepreau. Flexlab: A Realistic, Controlled, and Friendly Environment for Evaluating Networked Systems. In *Proc. HotNets V*, pages 103–108, Nov. 2006.

[10] E. Eide, L. Stoller, and J. Lepreau. An Experimentation Workbench for Replayable Networking Research. In *Proc. NSDI*, Apr. 2007.

[11] S. Floyd and E. Kohler. Internet Research Needs Better Models. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):29–34, Jan. 2003.

[12] S. Floyd and V. Paxson. Difficulties in Simulating the Internet. *IEEE/ACM TON*, 9(4):392–403, Aug. 2001.

[13] P. Francis, S. Jamin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang. IDMaps: A Global Internet Host Distance Estimation Service. *IEEE/ACM TON*, 9(5):525–540, Oct. 2001.

[14] M. Jain and C. Dovrolis. Ten Fallacies and Pitfalls on End-to-End Available Bandwidth Estimation. In *Proc. Conf. on Internet Measurement (IMC)*, pages 272–277, Oct. 2004.

[15] S. Jaiswal et al. Inferring TCP Connection Characteristics through Passive Measurements. In *Proc. INFOCOM*, pages 1582–1592, Mar. 2004.

[16] B. Krishnamurthy, H. V. Madhyastha, and O. Spatscheck. ATMEN: A Triggered Network Measurement Infrastructure. In *Proc. WWW*, pages 499–509, May 2005.

[17] A. Lakhina, M. Crovella, and C. Diot. Mining Anomalies Using Traffic Feature Distributions. In *Proc. SIGCOMM*, pages 217–228, Aug. 2005.

[18] T. V. Lakshman and U. Madhow. The Performance of TCP/IP for Networks with High Bandwidth-Delay Products and Random Loss. *IEEE/ACM TON*, 5(3):336–350, 1997.

[19] S.-J. Lee et al. Measuring Bandwidth Between PlanetLab Nodes. In *Proc. PAM*, pages 292–305, Mar.–Apr. 2005.

[20] X. Liu and A. Chien. Realistic Large-Scale Online Network Simulation. In *Proc. Supercomputing*, Nov. 2004.

[21] H. V. Madhyastha et al. iPlane: An Information Plane for Distributed Services. In *Proc. OSDI*, pages 367–380, Nov. 2006.

[22] A. Nakao, L. Peterson, and A. Bavier. A Routing Underlay for Overlay Networks. In *Proc. SIGCOMM*, pages 11–18, Aug. 2003.

[23] T. S. E. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *Proc. INFOCOM*, pages 170–179, June 2002.

[24] B. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proc. SIGCOMM*, pages 51–61, Sept. 1997.

[25] D. Oppenheimer, B. Chun, D. Patterson, A. C. Snoeren, and A. Vahdat. Service Placement in a Shared Wide-Area Platform. In *Proc. USENIX*, pages 273–288, May–June 2006.

[26] K. Park and V. Pai. CoMon: A Mostly-Scalable Monitoring System for PlanetLab. *ACM SIGOPS OSR*, 40(1):65–74, Jan. 2006.

[27] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):59–64, Jan. 2003.

[28] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM SIGCOMM CCR*, 27(1):31–41, Jan. 1997.

[29] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving Accuracy in End-to-end Packet Loss Measurement. In *Proc. SIGCOMM*, pages 157–168, Aug. 2005.

[30] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. SIGCOMM*, pages 133–145, Aug. 2002.

[31] N. Spring, L. Peterson, V. Pai, and A. Bavier. Using PlanetLab for Network Research: Myths, Realities, and Best Practices. *ACM SIGOPS OSR*, 40(1):17–24, Jan. 2006.

[32] N. Spring, D. Wetherall, and T. Anderson. Scriptroute: A Public Internet Measurement Facility. In *Proc. of USENIX USITS*, 2003.

[33] W. A. Taylor. Change-Point Analysis: A Powerful New Tool for Detecting Changes. http://www.variation.com/cpa/tech/changepoint.-html, Feb. 2000.

[34] A. Vahdat et al. Scalability and Accuracy in a Large-Scale Network Emulator. In *Proc. OSDI*, pages 271–284, Dec. 2002.

[35] A. Vahdat, L. Peterson, and T. Anderson. Public statements at PlanetLab workshops, 2004–2005.

[36] K. Webb, M. Hibler, R. Ricci, A. Clements, and J. Lepreau. Implementing the Emulab-PlanetLab Portal: Experience and Lessons Learned. In *Proc. WORLDS*, Dec. 2004.

[37] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[38] K. Xu, Z.-L. Zhang, and S. Bhattacharyya. Profiling Internet Backbone Traffic: Behavior Models and Applications. In *Proc. SIGCOMM*, pages 169–180, Aug. 2005.

[39] P. Yalagandula, P. Sharma, S. Banerjee, S.-J. Lee, and S. Basu. S3: A Scalable Sensing Service for Monitoring Large Networked Systems. In *Proc. SIGCOMM Workshop on Internet Network Mgmt. (INM)*, pages 71–76, Sept. 2006.

[40] M. Zhang et al. PlanetSeer: Internet Path Failure Monitoring and Characterization in Wide-Area Services. In *Proc. OSDI*, pages 167–182, Dec. 2004.

[41] Y. Zhang, N. Du, V. Paxson, and S. Shenker. On the Constancy of Internet Path Properties. In *Proc. SIGCOMM Internet Meas. Workshop (IMW)*, pages 197–211, Nov. 2001.

A Scheduling Accuracy

To quantify the jitter and delay in process scheduling on PlanetLab nodes, we implemented a test program that schedules a sleep with the `nanosleep()` system call, and measures the actual sleep time using `gettimeofday()`. We ran this test on three separate PlanetLab nodes with load averages of roughly 6, 15, and 27, plus an unloaded Emulab node running a PlanetLab-equivalent OS. 250,000 sleep events were continuously performed on each node with a target latency of 8 ms, for a total of about 40 minutes.

Figure 10 shows the CDF of the unexpected additional

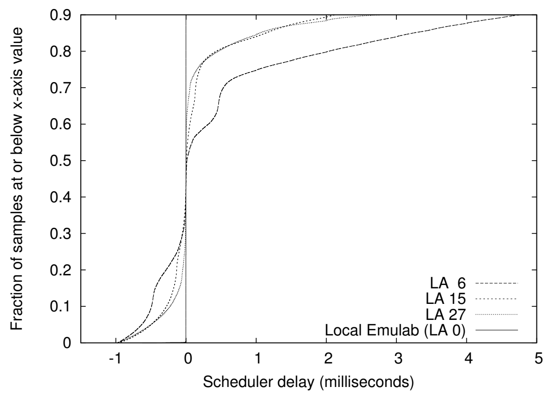


Figure 10: 90th percentile scheduling time difference CDF. The vertical line is "Local Emulab."

delay, up to the 90th percentile; Figure 11 displays the tail in log-log format. 90% of the events are within -1–5 scheduler quanta (msecs) of the target time. However, a significant tail extends to several hundred milliseconds. We also ran a one week survey of 330 nodes that showed the above samples to be representative.

This scheduling tail poses problems for the fidelity of programs that are time-sensitive. Many programs may still be able to obtain accurate results, but it is difficult to determine in advance which those are.

Spring et al. [31] also studied availability of CPU on PlanetLab, but measured it in aggregate instead of our timeliness-oriented measurement. That difference caused them to conclude that "PlanetLab has sufficient CPU capacity." They did document significant scheduling jitter in packet sends, but were concerned only with its impact on network measurement techniques. Our BT results strongly suggest that PlanetLab scheduling latency can greatly impact normal applications.

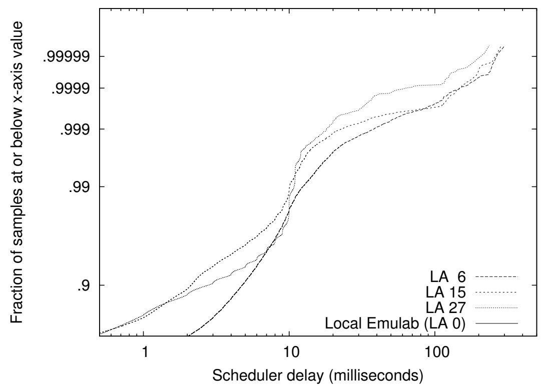


Figure 11: Log-log scale scheduling time difference CDF showing distribution tail. The "Local Emulab" line is vertical at $x = 0$.

An Experimentation Workbench for Replayable Networking Research

Eric Eide Leigh Stoller Jay Lepreau

University of Utah, School of Computing
{eeide, stoller, lepreau}@cs.utah.edu www.emulab.net

Abstract

The networked and distributed systems research communities have an increasing need for "replayable" research, but our current experimentation resources fall short of satisfying this need. Replayable activities are those that can be re-executed, either as-is or in modified form, yielding new results that can be compared to previous ones. Replayability requires complete records of experiment processes and data, of course, but it also requires facilities that allow those processes to actually be examined, repeated, modified, and reused.

We are now evolving Emulab, our popular network testbed management system, to be the basis of a new *experimentation workbench* in support of realistic, large-scale, replayable research. We have implemented a new model of testbed-based experiments that allows people to move forward and backward through their experimentation processes. Integrated tools help researchers manage their activities (both planned and unplanned), software artifacts, data, and analyses. We present the workbench, describe its implementation, and report how it has been used by early adopters. Our initial case studies highlight both the utility of the current workbench and additional usability challenges that must be addressed.

1 Introduction

In the networking and operating systems communities, there is an increasing awareness of the benefits of repeated research [5, 14]. A scientific community advances when its experiments are published, subjected to scrutiny, and repeated to determine the veracity of results. Repeated research not only helps to validate the conclusions of studies, but also to expand on previous conclusions and suggest new directions for research.

To repeat a piece of research, one first needs access to the complete records of the experiment that is to be redone. This obviously includes the results of the experiment—not only the final data products, but also the "raw" data products that are the bases for analysis. Data sets like those being collected in the networking community [2, 3, 6, 22] allow researchers to repeat analyses, but by themselves do not help researchers validate or repeat the data collection process. Therefore, the records of a repeatable experiment must also contain descriptions of the procedures that were followed, in sufficient detail to allow them to be re-executed. For studies of software-based systems, the documentation of an experiment should also contain copies of the actual software that was executed, test scripts, and so on.

A second requirement for repeated research is access to experimental infrastructure, i.e., laboratories. In the networking community, this need is being served by a variety of network testbeds: environments that provide resources for scalable and "real-world" experimentation. Some testbeds (such as Emulab [26]) focus on providing high degrees of control and repeatability, whereas others (such as PlanetLab [19]) focus on exposing networked systems to actual Internet conditions and users. Network testbeds differ from each other in many ways, but most have the same primary goal: to provide experimenters with access to resources. Once a person has obtained these resources, he or she usually receives little help from the testbed in actually *performing* an experiment: i.e., configuring it, executing it, and collecting data from it. Current testbeds offer few features to help users repeat research in practice. Moreover, they provide little guidance toward making new experiments repeatable.

Based on our experience in running and using Emulab, we believe that new *testbed-integrated, user-centered tools* will be a necessary third requirement for establishing repeatable research within the networked systems community. Emulab is our large and continually growing testbed: it provides access to many hundred computing devices of diverse types, in conjunction with user services such as file storage, file distribution, and user-scheduled events. As Emulab has grown over the past six years, its users have performed increasingly large-scale and sophisticated studies. An essential part of these activities is managing the many parts of every experiment, and we have seen first-hand that this can be a heavy load for Emulab's users. As both administrators and users of our testbed, we recognize that network researchers need better ways to organize, execute, record, and analyze their work.

In this paper we present our evolving solution: an integrated experiment management system called the *experimentation workbench*. The workbench is based on a new model of testbed-based experiments, one designed to describe the relationships between multiple parts of experiments and their evolution over time. The workbench enhances Emulab's previous model and existing features in order to help researchers "package" their experiment definitions, explore variations, capture experiment inputs and outputs, and perform data analyses.

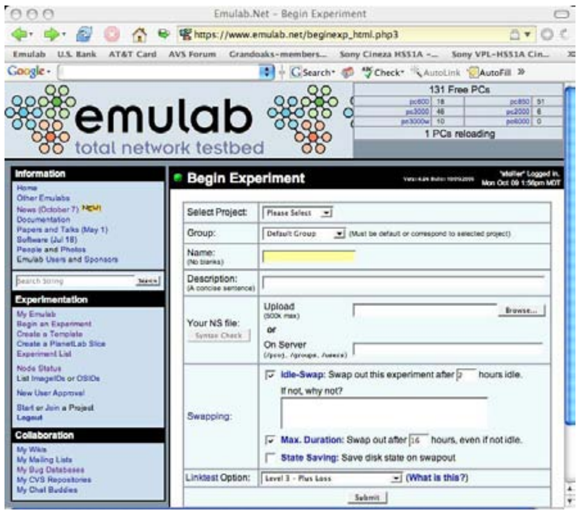
A key concern of the workbench is automation: we intend for users to be able to re-execute testbed-based experiments with minimum effort, either as-is or in modified form. Repeated research requires both experiment encapsulation and access to a laboratory. The workbench combines these things, encompassing both an experiment management facility *and* an experiment execution facility. This is in contrast to typical scientific workflow management systems [27], which automate processes but do not manage the "laboratories" in which those processes operate. Thus, instead of saying that our workbench supports repeated research, we say that it supports *replayable research*: activities that not only can be re-executed, but that are based on a framework that includes the resources necessary for re-execution.

The primary contributions of this paper are (1) the identification of *replayable research* as a critical part of future advances in networked systems; (2) the detailed presentation of our *experimentation workbench*, which is our evolving framework for replayable research; and (3) an evaluation of the current workbench through *case studies of its use* in actual research projects. Our workbench is implemented atop Emulab, but the idea of replayable research is general and applicable to other testbed substrates. In fact, two of our case studies utilize PlanetLab via the Emulab-PlanetLab portal [25].

This paper builds on our previous work [8] by detailing the actual workbench we have built, both at the conceptual level (Section 3) and the implementation level (Section 4). Our case studies (Section 5) show how the current workbench has been applied to ongoing network research activities, including software development and performance evaluation, within our research group. The case studies highlight both the usefulness of the current workbench and ways in which the current workbench should be improved (Section 5 and Section 6).

2 Background

Since April 2000, our research group has continuously developed and operated *Emulab* [26], a highly successful and general-purpose testbed facility and "operating system" for networked and distributed system experimentation. Emulab provides integrated, Web-based access to



Select Project:	Please Select
Group:	Default Group (Must be default or correspond to selected project)
Name: (No blanks)	
Description: (A concise sentence)	
Your NS file:	Upload (500k max) or On Server (/proj, /groups, /users)
Swapping:	[✓] Idle-Swap: Swap out this experiment after 2 hours idle. If not, why not? [✓] Max. Duration: Swap out after 16 hours, even if not idle. [ ] State Saving: Save disk state on swapout
Linktest Option:	Level 3 - Plus Loss (What is this?)

Figure 1: Emulab's Web interface

a wide range of environments including simulated, emulated, and wide-area network resources. It is a central resource in the network research and education communities: as of October 2006, the Utah Emulab site had over 1,500 users from more than 225 institutions around the globe, and these users ran over 18,000 experiments in the preceding 12 months. In addition to our testbed site at Utah, Emulab's software today operates more than a dozen other testbeds around the world.

The primary interface to Emulab is through the Web, as shown in Figure 1. Once a user logs in, he or she can start an *experiment*, which is Emulab's central unit of operation. An experiment defines both a static and a dynamic configuration of a network as outlined in Figure 2. Experiments are usually described in an extended version of the *ns* language [11], but they may also be described through a GUI within Emulab's Web interface.

The static portion describes a network topology: a set of devices (nodes), the network in which they are contained, and the configurations of those devices and network links. The description includes the type each node (e.g., a 3 GHz PC, a PlanetLab node, or a virtual machine), the operating system and other packages that are to be loaded onto each node, the characteristics of each network link, and so on. It also includes the definitions of *program agents*, which are testbed-managed entities that run programs as part of an experiment.

The dynamic portion is a description of events: activities that are scheduled to occur when the experiment is executed. An event may be scheduled for a particular time, e.g., thirty seconds after the start of the experiment. An event may also be unscheduled. In this case, the user or a running process may signal the event at run time. Events can be assembled into event sequences as shown

```
set ns [new Simulator]
source tb_compat.tcl

# STATIC PART: nodes, networks, and agents.
set cnode  [$ns node] # Define a node
set snode  [$ns node]
set lan    [$ns make-lan "$cnode $snode" 100Mb 0ms]
set client [$cnode program-agent]
set server [$snode program-agent]

# DYNAMIC PART: events.
set do_client [$ns event-sequence {
    $client run -command "setup.sh"
    $client run -command "client.sh"
}]
set do_server [$ns event-sequence {
    $server run -command "server.sh"
}]
set do_expt [$ns event-sequence {
    $do_server start  # Do not wait for completion
    $do_client run    # Run client, wait till end
}]
$ns at 0.0 "$do_expt run"
$ns run
```

Figure 2: A sample experiment definition

in Figure 2. Each event in a sequence is issued when the preceding activity completes; some activities (like `start`) complete immediately whereas others (like `run` to completion) take time. Events are managed and distributed by Emulab, and are received by various testbed-managed agents. Some agents are set up automatically by Emulab, including those that operate on nodes and links (e.g., to bring them up or down). Others are set up by a user as part of an experiment. These include the program agents mentioned above; *traffic generators*, which produce various types of network traffic; and *timelines*, which signal user-specified events on a timed schedule.

Through Emulab's Web interface, a user can submit an experiment definition and give it a name. Emulab parses the specification and stores the experiment in its database. The user can now "swap in" the experiment, meaning that it is mapped onto physical resources in the testbed. Nodes and network links are allocated and configured, program agents are created, and so on. When swap in is complete, the user can login to the allocated machines and do his or her work. A central NFS file server in Emulab provides persistent storage; this is available to all the machines within an experiment. Some users carry out their experiments "by hand," whereas others use events to automate and coordinate their activities. When the user is done, he or she tells Emulab to "swap out" the experiment, which releases the experiment's resources. The experiment itself remains in Emulab's database so it can be swapped in again later.

3 New Model of Experimentation

Over time, as Emulab was used for increasingly complex and large-scale research activities, we realized that the model of experiments described above fails to capture important aspects of the experimentation process.

3.1 Problems

We identified six key ways that the original Emulab model of experiments breaks down for users.

1. An experiment entangles the notions of definition and instance. An experiment combines the idea of describing a network with the idea of allocating physical resources for that description. This means, for example, that a single experiment description cannot be used to create two concurrent instances of that experiment.

2. The old model cannot describe related experiments, but representing such relationships is important in practice. Because distributed systems have many variables, a careful study requires running multiple, related experiments that cover a parameter space.

3. An experiment does not capture the fact that a single "session" may encompass multiple subparts, such as individual tests or trials. These subparts may or may not be independent of each other. For example, a test activity may depend on the prior execution of a setup activity. In contrast to item 2 about relating experiments to one another, the issue here is relating user sessions to (possibly many) experiments and/or discrete tasks.

4. Data management is not handled as a first-class concern. Users must instrument their systems under study and orchestrate the collection of data. For a large system with many high-frequency probes, the amount of data gathered can obviously be very large. Moreover, users need to analyze all their data: not just within one experiment, but also across experiments.

5. The old model does not help users manage *all the parts* of an experiment. In practice, an experiment is not defined just by an *ns* file, but also by all the software, input data, configuration parameters, and so on that is utilized within the experiment. In contrast to item 3 about identifying multiple "units of work," here we are concerned about collecting the many components of an experiment definition.

6. The old model does not help users manage their studies over time. The artifacts and purpose of experimentation may change in both planned and unplanned ways over the course of a study, which may span a long period of time. Saving and recalling history is essential for many purposes including collaboration, reuse, replaying experiments, and reproducing results.

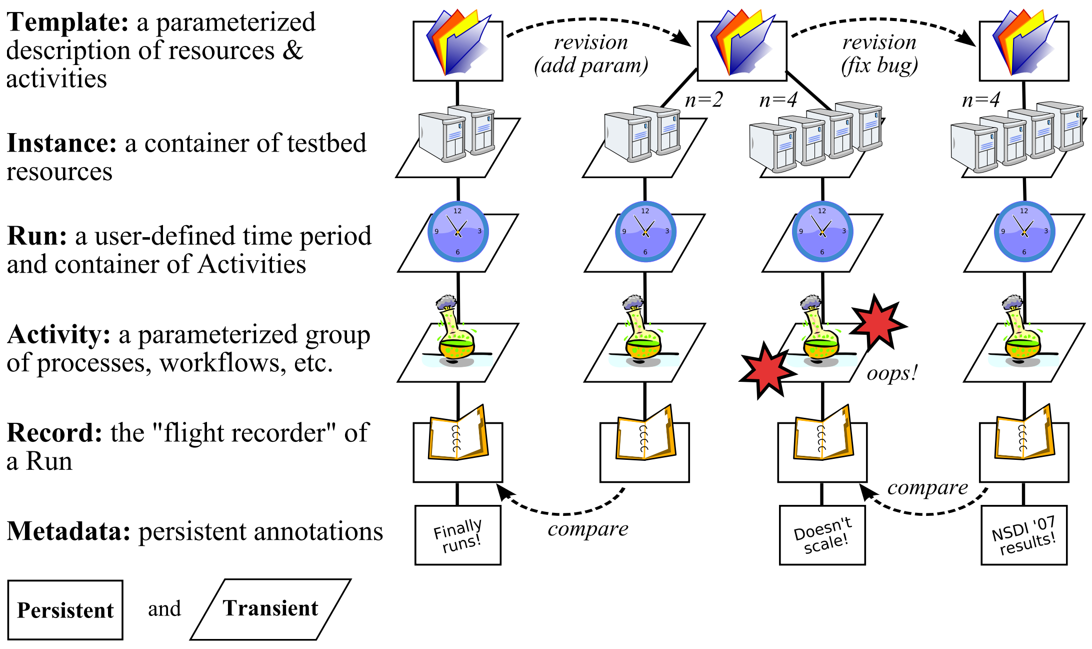

Template: a parameterized description of resources & activities
Instance: a container of testbed resources
Run: a user-defined time period and container of Activities
Activity: a parameterized group of processes, workflows, etc.
Record: the "flight recorder" of a Run
Metadata: persistent annotations
Persistent
and
Transient
revision (add param)
revision (fix bug)
n=2
n=4
n=4
oops!
compare
compare
Finally runs!
Doesn't scale!
NSDI '07 results!

Figure 3: Summary of the workbench model of experimentation. The diagram illustrates an example of a user's experimentation over time. (The information is displayed differently in the workbench GUI, as shown in other figures.) The user first creates a template and uses it to get one set of results, represented by the leftmost record. He or she then modifies the template to define a parameter. The new template is used to create two instances, and the activity in the four-node instance fails. The experimenter fixes the bug and creates a final template instance, in which the activity succeeds.

3.2 Solution: refine the model

The problems above deal with the key concerns of replayable research. Addressing the shortcomings of Emulab's experiment model, therefore, was an essential first step in the design and implementation of the workbench.

Our solution was to design a new, expanded model of testbed-based experiments. The original model of Emulab experiments is monolithic in the sense that a single user-visible entity represents the entirety of an experiment. This notion nevertheless fails to capture all the aspects of an experiment. Our new model divides the original notion of an experiment into parts, and then enhances those parts with new capabilities.

Figure 3 illustrates the main relationships between the components of the new model. The two most important components are *templates*, shown at the top, and *records*, shown at the bottom. These components are persistent and are stored "forever" by the workbench. In contrast, most of the other model components are transient because they represent entities that exist only while testbed resources are being used.

The rest of this section describes the elements of our new model at a high level. We have implemented the model (Section 4) and initial user experiences have been promising (Section 5), but they have also shown ways in which the model should be further refined (Section 6).

• A **template** is a repository for the many things that collectively define a testbed-based environment. It plays the "definition" role of Emulab's original experiment abstraction. Unlike an experiment, however, a template contains the many files that are needed for a study—not just an *ns* file, but also the source code and/or binaries of the system under test, input files, and so on. A template may also contain other kinds data such as (reified) database tables and references to external persistent storage, e.g., datapositories [3] and CVS repositories.

Templates have two additional important properties. First, templates are *persistent* and *versioned*. A template is an immutable object: "editing" a template actually creates a new template, and the many revisions of a template form a tree that a user can navigate. Second, templates have *parameters*, akin to the parameters to a function. Parameters allow a template to describe a family of related environments and activities, not just one.

• A **template instance** is a container of testbed resources: nodes, links, and so on. A user creates a template instance and populates it with resources by using the workbench's Web interface, which is an extension to Emulab's Web interface. The process of instantiating a template is very much like the process of swapping in an experiment in the traditional Emulab Web interface. There are two obvious differences, however. First, a user can specify parameter values when a template is instantiated. These values are accessible to processes within the instance and are included in records as described below. Second, a user can instantiate a template even if there is an existing instance that is associated with the template.

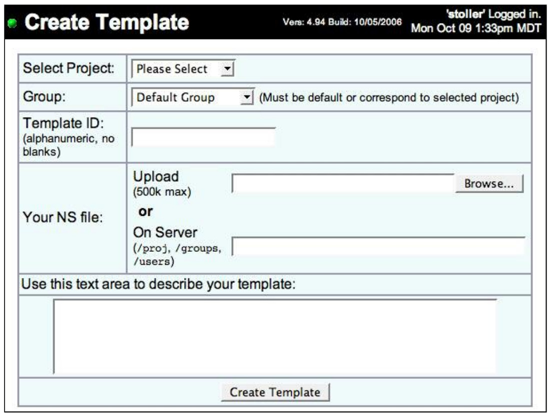
A template instance is a transient entity: it plays the “resource owner” role of the original, monolithic experiment notion. The resources within an instance may change over time, as directed by the activities that occur within it. When the activities are finished, the allocated resources are released and the instance is destroyed.

• A **run** is a user-defined context for activities. Conceptually, it is a container of processes that execute and events that occur within a template instance. In terms of the monolithic Emulab experiment model, the role of a run is to represent a user-defined “unit of work.” In the new model, a user can demarcate separate groups of activities that might occur within a single template instance. For example, a user could separate the individual trials of a system under test; these trials could occur serially or in parallel. (Our current implementation supports only serial runs.) A run is transient, but the events that occur within the run, along with the effects of those events (e.g., output files), are recorded as described below. Like a template instance, a run can have parameter values, which are specified when the run is created.

• An **activity** is a collection of processes, workflows, scripts, and so on that execute within a run. Having an explicit model component for activities is useful for two reasons. First, it is necessary for tracking the provenance of artifacts (e.g., output files) and presenting that provenance to experimenters. Second, it is needed for the workbench to manipulate activities in rich ways. For example, the workbench could execute only the portions of a workflow that are relevant to a particular output that a researcher wants to generate. In our current implementation, activities correspond to Emulab events and event sequences, plus the actions that are taken by testbed agents when events are received.

• A **record** is the persistent account of the activities and effects that occurred within a run. A record is a repository: it contains output files, of course, but it may also contain input files when those files are not contained within the template that is associated with the record. The idea of a record is to be a “flight recorder,” capturing everything that is relevant to the experimenter, at a user-specified level of detail. Once the record for a run is complete, it is immutable.

The workbench automatically captures data from well-defined sources, and special observers such as packet recorders [9], filesystem monitors [8], and provenance-aware storage systems [15] can help determine the “extent” of a testbed-based activity. These can be automatically deployed by a testbed to drastically lower the experiment-specification burden for users.


Create Template Vers: 4.94 Build: 10/05/2006 'stoller' Logged in. Mon Oct 09 1:33pm MDT

Select Project:	Please Select
Group:	Default Group (Must be default or correspond to selected project)
Template ID: (alphanumeric, no blanks)	
Your NS file:	Upload (500k max) Browse... or On Server (/proj, /groups, /users)
Use this text area to describe your template:	
	Create Template

Figure 4: Creating a new template

• Finally, **metadata** are used to annotate templates and records. Metadata are first-class objects, as opposed to being contained within other objects. This is important because templates and records are immutable once they are created. Through metadata, both users and the workbench itself can attach meaningful names, descriptions, and other mutable properties to templates and records.

4 Using the Workbench

In this section we present our current implementation of the workbench through an overview of its use. Our implementation extends Emulab to support replayable research, based on the conceptual model described in Section 3. Taken all at once, the model may seem overwhelming to users. An important part of our work, therefore, is to implement the model through GUI extensions and other tools that build upon the interfaces that testbed users are already accustomed to.

4.1 Creating templates

A user of the workbench begins by creating a new template. He or she logs in to Emulab Web site and navigates to the form for defining a new template, shown in Figure 4. The form is similar to the page for creating regular Emulab “experiments,” except that the controls related to swapping in an experiment are missing. The form asks the user to specify:

- the *project* and *group* for the template. These attributes relate to Emulab’s security model for users, described elsewhere [26].
- a *template ID*, which is a user-friendly name for the template, and an initial *description* of the template. These are two initial pieces of metadata.
- an *ns* file, which describes a network topology and a set of events, as described in Section 2.

Parameters. Templates may have parameters, and these are specified through new syntax in the *ns* file. A parameter is defined by a new *ns* command:

```
$ns define-template-parameter name value desc
```

where *name* is the parameter name, *value* is the default value, and *desc* is an optional descriptive string. These will be presented to the user when he or she instantiates the template. A parameter defines a variable in the *ns* file, so it may affect the configuration of a template instance.

Datastore. At this point, the user can click the *Create Template* button. The *ns* file is parsed, and the template is added to the workbench's database of templates. Now the user can add other files to the template. (In the future, we will extend the workbench GUI so that users can add files to a template as part of the initial creation step.)

When a template is created, the workbench automatically creates a directory representing the template at a well-known place in Emulab's filesystem. One can think of this as a "checkout" of the template from the repository that is kept by the workbench. The part of the template that contains files is called the *datastore*, and to put files into a template, a user places those files in the template's `datastore` directory.

```
# Navigate to the datastore of the template.
cd .../datastore
# Add scripts, files, etc. to the template.
cp ~/client.sh ~/server.sh .
cp -r ~/input-files .
```

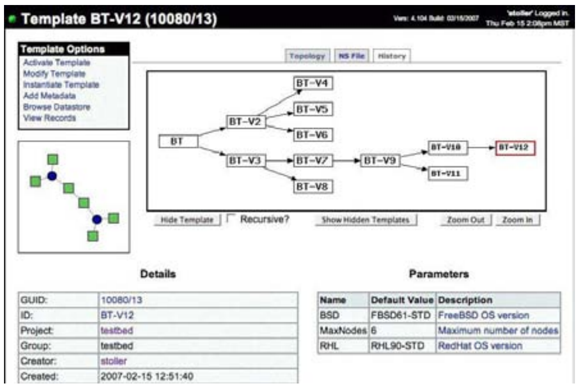
The user then "commits" the new files to the template.

```
template_commit
```

In fact, this action creates a *new template*. Recall that templates are immutable, which allows the workbench to keep track of history. Thus, a commit results in a new template, and the workbench records that the new template is derived from the original. The `template_commit` command infers the identity of the original template from the current working directory.

Our current implementation is based on Subversion, the popular open-source configuration management and version control system. This is hidden from users, however: the directories corresponding to a template are not a "live" Subversion sandbox. So far, we have implemented the ability to put ordinary files into a template. Connecting a template to other sources of persistent data, such as databases or external source repositories, is future implementation work. Emulab already integrates CVS (and soon Subversion) support for users, so we expect to connect the workbench to those facilities first.

Template history. A template can be "modified" either through the filesystem or through the workbench's



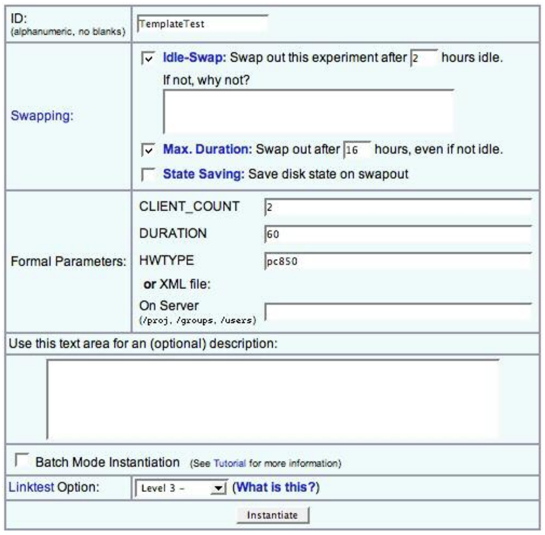
Details

GUID:	10080/13
ID:	BT-V12
Project:	testbed
Group:	testbed
Creator:	stoller
Created:	2007-02-15 12:51:40

Parameters

Name	Default Value	Description
BSD	FBSD61-STD	FreeBSD OS version
MaxNodes	6	Maximum number of nodes
RHL	RHL90-STD	RedHat OS version

Figure 5: The workbench tracks and displays how templates are derived



ID: (alphanumeric, no blanks)	TemplateTest
Swapping:	✓ Idle-Swap: Swap out this experiment after 2 hours idle. If not, why not? ✓ Max. Duration: Swap out after 16 hours, even if not idle. ☐ State Saving: Save disk state on swapout
Formal Parameters:	CLIENT_COUNT 2; DURATION 60; HWTYPE pc850; or XML file: On Server (/proj, /groups, /users)
Use this text area for an (optional) description:	
☐ Batch Mode Instantiation (See Tutorial for more information)	
Linktest Option:	Level 3 - (What is this?)

Figure 6: Instantiating a template

Web interface. Each modification results in a new template, and a single template may be modified multiple times—for example, to explore different directions of research. Thus, over a time, a user creates a tree of templates representing the history of a study. The workbench tracks this history and can present it to users as shown in Figure 5. The original template is the root of the tree, at left; the most recent versions of the template are at the right. By clicking on the nodes of the tree, a user can recall and inspect any template in the history. The workbench also provides controls that affect the tree display: e.g., individual nodes or subtrees can be elided.

4.2 Instantiating templates

Through the Web, a user can select a template and instantiate it. The form for instantiating a template is shown in Figure 6; it is similar to the form for swapping in a regular Emulab experiment, with two primary differences. First, the template form displays the template parameters and lets the user edit their values. Second, the user must give a name to the template instance. This name is used by Emulab to create DNS records for the machines that are allocated to the template instance.

When a template is instantiated, a new copy of the template's datastore is created for the instance. This ensures that concurrent instances of a template will not interfere with each other through modifications to files from the datastore. This feature also builds on existing Emulab practices. Users already know that Emulab creates a directory for each experiment they run; the workbench extends this by adding template-specific items, such as a copy of the datastore, to that directory.

4.3 Defining activities

When a template instance is created, testbed resources are allocated, configured, and booted. After the network and devices are up and running, the workbench automatically starts a *run* (Section 3.2) and starts any prescheduled activities within that run. The parameters that were specified for the template instance are communicated to the agents within the run.

Predefined activities. In our current implementation, activities are implemented using events and agents. Agents are part of the infrastructure provided by Emulab; they respond to events and perform actions such as modifying the characteristics of links or running user-specified programs. We found that the existing agent and event model was well suited to describing "pre-scripted" activities within our initial workbench.

Agents and scripted events are specified in a template's *ns* file, and commonly, events refer to data that is external to the *ns* file. For example, as illustrated in Figure 2, events for program agents typically refer to external scripts. The workbench makes it possible to encapsulate these files within templates, by putting them in a template's datastore. When a template is instantiated, the location of the instance's copy of the datastore is made available via the DATASTORE variable. An experimenter can use that to refer to files that are contained within the template, as shown in this example:

```
set do_client [$ns event-sequence {
  $client run -command {$DATASTORE/setup.sh}
  $client run -command {$DATASTORE/client.sh}
}]
```

Dynamically recording activities. The workbench also allows a user to record events dynamically, for replay in the future. This feature is important for lowering the barrier to entry for the workbench and supporting multiple modes of use. To record a dynamic event, a user executes the `template_record` command on some host within the template instance. For example, the following records an event to execute `client.sh`:

```
template_record client.sh
```

A second instance of `template_record` adds a second event to the recording, and so on. The workbench provides additional commands allow a user to stop and restart time with respect to the dynamic record, so that the recording does not contain large pauses when it is later replayed. When a recording is complete, it can be edited using a simple Web-based editor.

4.4 Using records

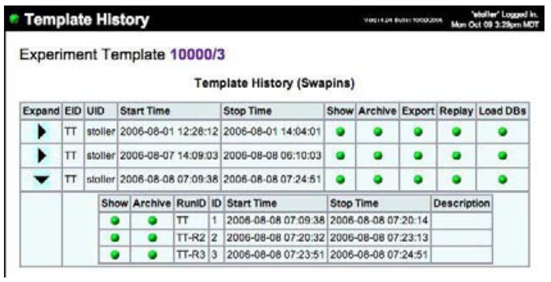
A record is the flight recorder of all the activities and effects that occur during the lifetime of a run. To achieve this goal in a transparent way, the workbench needs to fully instrument the resources and agents that constitute a template instance. We are gradually adding such instrumentation to the testbed, and in the meantime, we use a combination of automatic and manual (user-directed) techniques to decide what should be placed in a record.

Creating records. When a run is complete—e.g., because the experimenter uses the Web interface to terminate the template instance—the workbench creates a record of the run containing the following things:

- *the parameter values* that were passed from the template instance to the run.
- *the logs* that were generated by testbed agents. These are written to well-known places, so the workbench can collect them automatically.
- *files* that were written to a special `archive` directory. Similar to the `datastore` directory described above, every template instance also has an `archive` directory in which users can place files that should be persisted.
- *the recorded dynamic events*, explained above.
- *a dump of the database* for the template instance. Similar to the `archive` directory, the workbench automatically creates an online database as part of every template instance. The activities within a template instance can use this database as they see fit.

As described for templates, records are also stored in a Subversion repository that is internal to the workbench. We chose this design to save storage space, since we anticipated that the different records derived from a single template would be largely similar. Our experience, however, is that Subversion can be too slow for our needs when it is asked to process large data sets. We describe our experience with Subversion further in Section 6.

Inspecting records. The workbench stores records automatically and makes them available to users through



Template History

'stoller' Logged In. Mon Oct 09 3:29pm MDT

Experiment Template 10000/3

Template History (Swapins)

Expand	EID	UID	Start Time	Stop Time	Show	Archive	Export	Replay	Load DBs
▶	TT	stoller	2006-08-01 12:28:12	2006-08-01 14:04:01	●	●	●	●	●
▶	TT	stoller	2006-08-07 14:09:03	2006-08-08 06:10:03	●	●	●	●	●
▼	TT	stoller	2006-08-08 07:09:38	2006-08-08 07:24:51	●	●	●	●	●

Show	Archive	RunID	ID	Start Time	Stop Time	Description
●	●	TT	1	2006-08-08 07:09:38	2006-08-08 07:20:14	
●	●	TT-R2	2	2006-08-08 07:20:32	2006-08-08 07:23:13	
●	●	TT-R3	3	2006-08-08 07:23:51	2006-08-08 07:24:51	

Figure 7: Viewing the records associated with a template

the Web. From the Web page that describes a template, a user can click on the *View Records* link to access the records that have been derived from that template. Figure 7 shows an example. Records are arranged in a two level hierarchy corresponding to template instances and runs within those instances. A user can navigate to any record and inspect its contents. Additional controls allow a user to reconstitute the database that was dumped to a record: this is useful for post-mortem analysis, either by hand or in the context of another run. Finally, the workbench provides a command-line program for exporting the full contents of a record.

Replay from records. The Web interface in Figure 7 includes a *Replay* button, which creates a new template instance from a record. The replay button allows a user to create an instance using the parameters and datastore contents that were used in the original run. The original *ns* file and datastore are retrieved from the template and/or record, and the parameter values come from the record. A new instance is created and replayed, eventually producing a new record of its own.

4.5 Managing runs

A user may choose to enclose all of his or her activities within a single run. Alternatively, a user may start and stop multiple runs during the lifetime of a template instance, thus yielding multiple records for his or her activities. Whenever a new run begins, the user may specify new parameter values for that run. The set of parameters are those defined by the template; the user simply has an opportunity to change their values for the new run.

When a run stops, the workbench stops all the program agents and tracing agents within the run; collects the log files from the agents; dumps the template instance's database; and commits the contents of the instance's `archive` directory to the record. When a new run begins, the workbench optionally cleans the agent logs; optionally resets the instance's database; communicates new parameter values to the program agents; restarts the program agents; and restarts "event time" so that scheduled events in the template's *ns* file will reoccur.

In addition to starting and stopping runs interactively, users can script sequences of runs using the event system and/or Emulab's XML-RPC interface. Through the event system, an experimenter can use either static (i.e., scheduled) or dynamic events to control runs. Using a command-line program provided by Emulab—or one written by the user in any language that supports XML-RPC—a user can start and stop runs programmatically, providing new parameter values via XML.

5 Case Studies

The experimentation workbench is a work in progress, and continual feedback from Emulab users is essential if we are to maximize the applicability and utility of the workbench overall. We therefore recruited several members of our own research group to use our prototype in late 2006. All were experienced testbed users (and developers), but not directly involved in the design and implementation of the workbench. They used the workbench for about a month for their own research in networked systems, as described below.

5.1 Study 1: system development

The first study applied the workbench to software development tasks within the Flexlab project [7]. Flexlab is software to support the emulation of "real world" network conditions within Emulab. Specifically, Flexlab emulates conditions observed in PlanetLab, an Internet-based overlay testbed, within Emulab. The goal of Flexlab is to make it possible for applications that are run within Emulab to be subject to the network conditions that would be present if those applications had been run on PlanetLab. The Flexlab developers used the workbench to improve the way in which they were developing and testing Flexlab itself.

A Flexlab configuration consists of several pairs of nodes, each pair containing one Emulab node and one "proxy" of that node in PlanetLab. The Flexlab infrastructure continually sends traffic between the PlanetLab nodes. It observes the resulting behavior, produces a model of the network, and directs Emulab to condition its network links to match the model. The details of these processes are complex and described elsewhere [7].

The original framework for a Flexlab experiment consisted of (1) an *ns* file, containing variables that control the topology and the specifics of a particular experiment, and (2) a set of scripts for launching the Flexlab services, monitoring application behavior, and collecting results. A Flexlab developer would start a typical experiment by modifying the *ns* file variables as needed, creating an Emulab experiment with the modified *ns* file, and running a "start trial" script to start the Flexlab infrastructure. Thus, although all the files were managed via CVS, each experiment required by-hand modification of the *ns* file, and the Emulab experiment was not strongly associated with the files in the developer's CVS sandbox.

At this point the experimenter would run additional scripts to launch an application atop Flexlab. When the application ended, he or she would run a "stop trial" script to tear down the Flexlab infrastructure and collect results. He or she would then analyze the results, and possibly repeat the start, stop, and analysis phases a number of times. The attributes of each trial could be changed either by specifying options to the scripts or, in some cases, by modifying the experiment (via Emulab's Web interface). Thus, although the collection of results was automated, the documentation and long-term archiving of these results were performed by hand—or not at all. In addition, the configuration of each trial was accomplished through script parameters, not through a testbed-monitored mechanism. Finally, if the user modified the experiment, the change was destructive. Going back to a previous configuration meant undoing changes by hand or starting over.

The Flexlab developers used the workbench to start addressing the problems described above with their ad hoc Flexlab testing framework. They created a template from their existing *ns* file, and it was straightforward for them to change the internal variables of their original *ns* file to be parameters of the new template. They moved the start- and stop-trial scripts into the template datastore, thus making them part of the template. The options passed to these scripts also became template parameters: this made it possible for the workbench to automatically record their values, and for experimenters to change those values at the start of each run (Section 4.5). Once the scripts and their options were elements of the template, it became possible to modify the template so that the start- and stop-trial scripts would be automatically triggered at run boundaries. The Flexlab developers also integrated the functions of assorted other maintenance scripts with the start-run and stop-run hooks. A final but important benefit of the conversion was that the workbench now performs the collection and archiving of result files from each Flexlab run.

These changes provided an immediate logistical benefit to the Flexlab developers. A typical experiment now consists of starting with the Flexlab template, setting the values for the basic parameters (e.g., number of nodes in the emulated topology, and whether or not PlanetLab nodes will be needed), instantiating the template, and then performing a series of runs via menu options in the workbench Web interface. Each run can be parameterized separately and given a name and description to identify results.

At the end of the case study, the Flexlab developers made four main comments about the workbench and their overall experience. First, they said that although the experience had yielded a benefit in the end, the initial fragility of the prototype workbench sometimes made it more painful and time-consuming to use than their "old" system. We fixed implementation bugs as they were reported. Second, the developers noted that a great deal of structuring had already been done in the old Flexlab environment that mirrors some of what the workbench provides. Although this can be seen as reinvention, we see it as a validation that the facilities of the workbench are needed for serious development efforts. As a result of the case study, the Flexlab developers can use the generic facilities provided by the workbench instead of maintaining their own ad hoc solutions. Third, the Flexlab developers noted that they are not yet taking advantage of other facilities that the workbench offers. For example, they are not recording application data into the per-instance database (which would support SQL-based analysis tools), nor are they using the ability to replay runs using data from previous records. Fourth, the developers observed that the prototype workbench could not cooperate with their existing source control system, CVS. Integrating with such facilities is part of our design (Section 3.2), but is not yet implemented.

Despite the current limits of the workbench, the Flexlab developers were able to use it for real tasks. For instance, while preparing their current paper [21], they used the workbench in a collaborative fashion to manage and review the data from dozens of experiments.

5.2 Study 2: performance analysis

We asked one of the Flexlab developers to use the workbench in a second case study, to compare the behavior of BitTorrent on Flexlab to the behavior of BitTorrent on PlanetLab.

He created templates to run BitTorrent configurations on both testbeds. His templates automated the process of preparing the network (e.g., distributing the BitTorrent software), running BitTorrent, producing a consolidated report from numerous log files, and creating graphs using `gnuplot`. Data were collected to the database that is set up by the workbench for each template instance, and the generated reports and graphs were placed into the record. These output files were then made available via Emulab's Web interface (as part of the record).

Once a run was over, to analyze collected data in depth, the developer used the workbench Web interface to reactivate the live database that was produced during the run. The performance results that the researcher obtained through workbench-based experiments—too lengthy to present here—are included in the previously cited NSDI '07 paper about Flexlab [21].

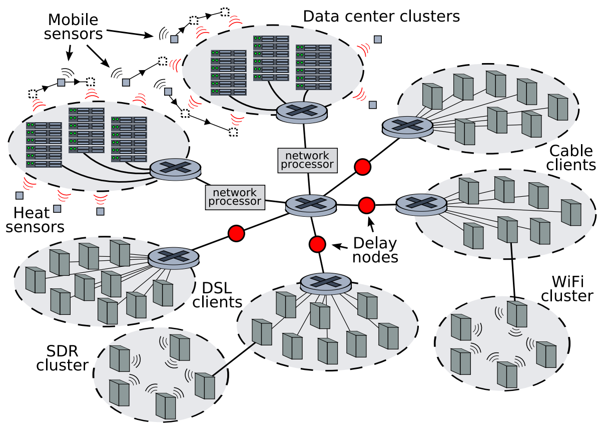


Figure 8: An example GHETE topology. GHETE utilizes the many types of emulated and actual devices in Utah's Emulab.

In terms of evaluating the workbench itself, the developer in this case study put the most stress on our prototype—and thereby illuminated important issues for future workbench improvements. For example, he initially had problems using the workbench in conjunction with PlanetLab, because our prototype workbench was not prepared to handle cases in which nodes become unavailable between runs. He also asked for new features, such as the ability to change the *ns* file during a template instance, and have those changes become "inputs" to subsequent runs. We had not previously considered the idea that a user might want to change not just parameter values, but the entire *ns* file, between runs in a single template instance. We quickly implemented support for these features, but we will clearly need to revisit the issues that were raised during this case study. In particular, it is clear that the issue of handling resource failures will be important for making the workbench robust and applicable to testbeds such as PlanetLab.

5.3 Study 3: application monitoring

Another local researcher used our workbench to study the behavior of an existing large-scale emulation scenario called GHETE (Giant HETerogeneous Experiment).

GHETE was written by the researcher in June 2006 to showcase Emulab's ability to handle large network topologies containing many types of nodes and links: wired and wireless PCs, virtual nodes, software-defined radio (SDR) nodes, sensor network motes, and mobile robots carrying motes. GHETE uses these node types inside an emulation of a distributed data center, such as the scenario shown in Figure 8. A data center consists of two (or more) clusters, each acting as a service center for large numbers of client nodes. Client nodes send large TCP streams of data to the cluster servers via `iperf`, emulating an Internet service such as a heavily used, distributed file backup service. Each cluster is monitored for excessive heat by a collection of fixed and mobile sensor nodes. When excessive heat is detected at a cluster, the cluster services are stopped, emulating a sudden shutdown. A load-balancing program monitors cluster bandwidth and routes new clients to the least-utilized cluster. The clients in Figure 8 are connected to the data center through emulated DSL connections, emulated cable modem links, and true 802.11 wireless and SDR networks.

The GHETE developer used the workbench template system to parameterize many aspects of the emulation, including client network size, bandwidth and latency characteristics, and node operating systems. Because the arguments controlling program execution were also turned into template parameters, the workbench version of GHETE allows an experimenter to execute multiple runs (Section 4.5) to quickly evaluate a variety of software configurations on a single emulated topology.

Although the workbench version of GHETE provided many avenues for exploration, we asked the developer to focus his analysis on the behavior of GHETE's load balancer. To perform this study, he defined a GHETE topology with two service clusters. The aggregate bandwidth of each cluster was measured at one-second intervals, and these measurements were stored in the database that was created by the workbench for the template instance. To visualize the effectiveness of the load balancer, the developer wrote a script for the R statistics system [20] that analyzes the collected bandwidth measurements. The script automates post-processing of the data: it executes SQL queries to process the measurements directly from the database tables and generates plots of the results.

Figure 9 and Figure 10 show two of the graphs that the developer produced from his automated activities during the study. Each shows the effect of a given load-balancing strategy by plotting the absolute difference between the aggregate incoming bandwidths of the two service clusters over time. The spikes in the graphs correspond to times when one cluster was "shut down" due to simulated heat events. The results show that the second load balancer (Figure 10) yields more stable behavior, but the details of the load balancers are not our focus here. Rather, we are interested in how the workbench was used: did it help the GHETE developer perform his experiments and analyses more efficiently?

The GHETE developer said that the workbench was extremely useful in evaluating the load balancers in GHETE and provided insights for future improvements. He listed three primary ways in which the workbench improved upon his previous development and testing methods. First, as noted above, by providing the opportunity to set new parameter values for each run within a template instance, software controlled by Emulab program agents

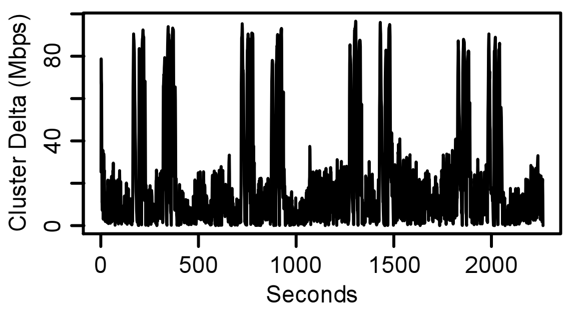


Figure 9: Measured difference in the incoming bandwidths of the two service clusters, using a simple load balancer

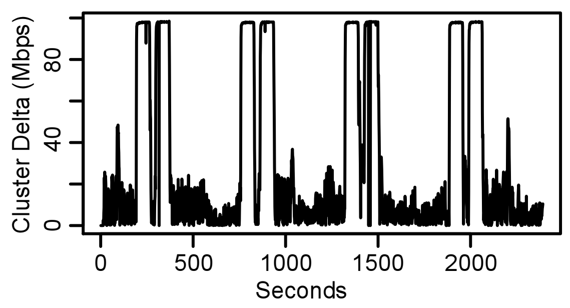


Figure 10: Measured difference in the incoming bandwidths of the two service clusters, using an improved load balancer

could be easily configured and reconfigured. This enabled rapid trials and reduced experimenter wait time. Second, via the per-template-instance database, the developer was able to analyze incoming data in real time within *interactive* R sessions. This is in addition to the post-processing analysis described above. He could also easily re-instantiate the database from any completed run for further analysis. Third, since he parameterized all relevant aspects of GHETE's emulation software, he was able to quickly iterate through many different parameters without wasting time tracking which runs correspond to which parameter settings. Since the parameter bindings for each run are archived by the workbench, he was able to easily recall settings in the analysis phase.

6 Lessons Learned

In this section, we summarize two categories of "lessons learned" from the case studies and our other experiences to date with the prototype workbench. First, we discuss storage issues and the effect of our philosophy to have the workbench "store everything" by default. Second, we evaluate how well the workbench's model of experimentation actually describes the processes and relationships that are important to testbed-based research.

	Record size (KB)	**Stored in repo. (KB)**	**Elapsed time (s)**
BitTorrent (§5.2)	31,782	19,380	418
GHETE (§5.3)	72,330	20,240	861
Minimal template	88	48	48

Table 1: Space and time consumed for typical records in our case studies. The first column shows the size of the record, and the second shows the size of that record in the workbench's repository. The third shows the time required to stop a run, collect the data from all nodes, and commit the record. "Minimal template" describes the overhead imposed by the workbench.

6.1 Storage issues

It is essential for the workbench to store templates and records efficiently, because we intentionally adopted design principles that lead to high storage demands: "complete" encapsulation of experiments, proactive experiment monitoring, and saving history "forever" [8]. As described in Section 4, our prototype uses Subversion to store both templates and records in a space-efficient manner. We expected that the templates within a history tree would be quite similar to each other, and thus would be efficiently stored as deltas within Subversion. This argument also applies to the many records that arise from a template, but perhaps to a lesser degree. It was surprising to us, therefore, when we realized that the two primary lessons we learned about workbench storage were not about space, but about time and clutter.

First, as reported by our users, the prototype is too slow at processing large amounts of data. This is most commonly seen at the end of a run, when the workbench collects data from many nodes and commits the assembled record. The current workbench forces a user to wait until a record is complete before he or she can safely resume experimentation within a template instance. Table 1 describes typical records created from the templates used in our second and third case studies.

Our conclusion is that the workbench must collect and persist data more efficiently *from the user's point of view.* This deals partly with the amount of data collected, but more significantly, it deals with when that data is gathered. Both issues can be addressed by overlapping record-making with user activities in time. For example, the workbench may leave records "open" for a while. Log collection would occur in parallel with other user activities, i.e., new runs. Collected data would be held in escrow, and users would have a window in which to add or remove items from a record. When the window closes, user-specified rules would apply, and the record would be finalized.

To isolate record-gathering and user activities that run in parallel, we are investigating the use of branching filesystems. In particular, we have enhanced the production-quality Linux “LVM” logical volume manager software [12]. Our extensions support the arbitrary branching of filesystem snapshots and greatly improve the performance of chained snapshots overall. A branching filesystem can prevent new runs from modifying the files created by previous runs. When deployed for the workbench repository (i.e., a shared filesystem), snapshots can naturally and efficiently encapsulate the contents of both templates and records. When deployed on the nodes within a template instance, it can remove the need for users to put all the files that they want saved into special directories for collection: all the files that are produced during a particular run can be identified as part of a particular snapshot. This “universal” archiving will raise issues of user intent, as sometimes users won’t wish certain files to be persisted and/or restored. Such cases can be dealt with as described above, by leaving records “open” and applying user-specified rules.

The second lesson we learned was that, despite our vision of a workbench that stores data forever, users have a strong desire to delete data. Our users continually asked for the ability to destroy templates and records. These requests were not motivated by storage space, since we had plenty available. Instead, our users wanted to reduce cognitive clutter and maintain their privacy. They did not want to see their “junk” forever, and neither did they like the idea that their “mistakes” and other private activities might be forever viewable by others.

Although the records of old and failed experiments can be valuable for many reasons, it is clear that we need to provide a range of deletion options within the workbench. These will range from merely hiding templates and records to expunging them forever. Both present a challenge in terms of presenting undeleted objects in a consistent and complete way, and the latter puts a new requirement on the workbench data repository.

6.2 Model issues

The workbench is based on the model of experimentation described in Section 3. Based on our experience as users and administrators of Emulab, we designed the model to express notions and relationships that we knew needed to be captured over the course of testbed-based research. Overall, our experience with the new model has been very positive. Its abstractions map well to mechanisms that experimenters design for themselves, as we saw in our first case study (Section 5.1). Many of the problems that our users encountered were due to the immaturity of our prototype—i.e., unimplemented features—as opposed to problems with the model.

On the other hand, the users in our first and second cases studies also had problems that *were* traceable to shortcomings of the model. In the first study, there was confusion about the relationships between runs: does a run start with the environment (e.g., filesystem and database) that was left by the preceding run, or does each run start afresh? In the second study, the experimenter wanted to substantially change his network topology between runs, and he had trouble with the workbench when nodes failed unexpectedly. All these issues stem from the *life cycle concerns* of objects and abstractions that are not clearly represented in our new model of experimentation. We can and will refine the model to address the issues that emerged during our case studies.

We always expected to refine our model as we gained experience with the workbench. What we learned is that many of the problems of our current model stem from a single key difference between the requirements of our workbench and the requirements of experiment management systems for other scientific domains. It is this: Unlike systems that track workflows and data within static laboratories, the workbench must manage a user’s activities *and* the user’s dynamic laboratory at the same time.

The user’s laboratory contains his or her testbed resources, which have dynamic state and behavior. The life cycles of the user’s laboratory and experimental activities are separate but intertwined. Our workbench models the separation—it distinguishes template instances from runs—but not the interactions that should occur between these two levels of testbed use. For example, it is not enough for template instances to start and stop; they should also be able to handle resource failures on their own, via user-specified procedures. Instances also need to communicate with runs to coordinate the handling of failures with users’ experimental activities. Some existing systems (e.g., Plush [1]) model concerns such as these, and their models will help to guide the evolution of our workbench.

7 Related Work

The workbench is an experiment management system, and there is a growing awareness of the need for such systems within the networking research community. Plush [1], for instance, is a framework for managing experiments in PlanetLab. A Plush user writes an XML file that describes the software that is to be run, the testbed resources that must be acquired, how the software is to be deployed onto testbed nodes, and how the running system is to be monitored. At run time, Plush provides a shell-like interface that helps a user perform resource acquisition and application control actions across testbed nodes. Plush thus provides features that are similar to those provided by Emulab’s core management software, which utilizes extended *ns* files and provides a Web-based GUI. Our workbench builds atop these services to address higher-level concerns of experimentation management: encapsulation and parameterization via templates, revision tracking and navigation, data archiving, data analysis, and user annotation. Plush and our work are therefore complementary, and it is conceivable that a future version of the workbench could manage the concerns mentioned above for Plush-driven experiments.

Weevil [24] is a second experiment management system that has been applied to PlanetLab-based research. Weevil is similar to Plush and Emulab's control system in that it deals with deploying and executing distributed applications on testbeds. It is also similar to the workbench in that it deals with parameterization and data collection concerns. Weevil is novel, however, in two primary ways. First, Weevil uses generative techniques to produce both testbed-level artifacts (e.g., topologies) and application-level artifacts (e.g., scripts) from a set of configuration values. Our workbench uses parameters to configure network topologies in a direct way, and it makes parameters available to running applications via program agents. It does not, however, use parameters to generate artifacts like application configuration files, although users can automate such tasks for themselves via program agents. Weevil's second novel feature is that it places a strong focus on workload generation as part of an experiment configuration. Both of the features described above would be excellent additions to future versions of the workbench. As with Plush, Weevil and the workbench are largely complementary because they address different concerns of replayable research.

Many scientific workflow management systems have been developed for computational science, including Kepler [13], Taverna [17], Triana [23], VisTrails [4], and others [27]. Many of these are designed for executing distributed tasks in the Grid. Our workbench has much in common with these systems in that the benefits of workflow management include task definition and annotation, tracking data products, and promoting exploration and automation. Our workbench differs from general scientific workflow management systems, however, in terms of its intended modes of use and focus on networked systems research. Our experience is that Emulab is most successful when it does not require special actions from its users; the workbench is therefore designed to enhance the use of existing testbeds, not to define a new environment. Because the workbench is integrated with a network testbed's user interface, resource allocators, and automation facilities, it can do "better" than Grid workflow systems for experiments in networked systems.

Many experiment management systems have also been developed for domains outside of computer science. For instance, ZOO [10] is a generic management environment that is designed to be customized for research in fields such as soil science and biochemistry. ZOO is designed to run simulators of physical processes and focuses primarily on data management and exploration. Another experiment management system is LabVIEW [16], a popular commercial product that interfaces with many scientific instruments. It has been used as the basis of several "virtual laboratories," and like Emulab, LabVIEW can help users manage both real and simulated devices. Our experimentation workbench is a significant step toward bringing the benefits of experiment management, which are well known in the hard sciences, to the domain of computer science in general and networked and distributed systems research in particular. Networking research presents new challenges for experiment management: for instance, the "instruments" in a network testbed consume and produce many complex types of data including software, input and output files, and databases of results from previous experiments. Networking also presents new opportunities, such as the power of testbeds with integrated experiment management systems to reproduce experiments "exactly" and perform new experiments automatically. Thus, whereas physical scientists must be satisfied with repeatable research, we believe that the goal of computer scientists should be *replayable research*: encapsulated activities *plus* experiment management systems that help people re-execute those activities with minimum effort.

8 Conclusion

Vern Paxson described the problems faced by someone who needs to reproduce his or her own network experiment after a prolonged break [18]: "It is at this point—we know personally from repeated, painful experience—that trouble can begin, because the reality is that for a complex measurement study, the researcher will often discover that they cannot reproduce the original findings precisely! The main reason this happens is that the researcher has now lost the rich mental context they developed during the earlier intense data-analysis period." Our goal in building an experimentation workbench for replayable research is to help researchers overcome such barriers—not just for re-examining their own work, but for building on the work of others.

In this paper we have forwarded the idea of *replayable research*, which pairs repeatable experiments with the testbed facilities that are needed to repeat and modify experiments in practice. We have presented the design and implementation of our experimentation workbench that supports replayable research for networked systems, and we have described how early adopters are applying the evolving workbench to actual research projects. Our new model of testbed-based experiments is applicable to network testbeds in general; our implementation extends the Emulab testbed with new capabilities for experiment management. The workbench incorporates and helps to automate the community practices that Paxson suggests [18]: e.g., strong data management, version control, encompassing "laboratory notebooks," and the publication of measurement data. Our goal is to unite these practices with the testbed facilities that are required to actually replay and extend experiments, and thereby advance *science* within the networking and distributed systems communities.

Acknowledgments

We gratefully thank Kevin Atkinson, Russ Fish, Sachin Goyal, Mike Hibler, David Johnson, and Robert Ricci for their participation in the case studies. Tim Stack did much early implementation for the workbench, and several other members of our group provided feedback on its current design. Juliana Freire, Mike Hibler, and Robert Ricci all made significant contributions to the workbench design and user interface. We thank David Andersen for his work on the datapository and Prashanth Radhakrishnan for his work on branching LVM. Finally, we thank the anonymous NSDI reviewers and our shepherd, Rebecca Isaacs, for their many constructive comments.

This material is based upon work supported by NSF under grants CNS–0524096, CNS–0335296, and CNS–0205702.

References

[1] J. Albrecht, C. Tuttle, A. C. Snoeren, and A. Vahdat. PlanetLab application management using Plush. *ACM SIGOPS OSR*, 40(1):33–40, Jan. 2006.

[2] M. Allman, E. Blanton, and W. Eddy. A scalable system for sharing Internet measurements. In *Proc. Passive and Active Measurement Workshop (PAM)*, Mar. 2002.

[3] D. G. Andersen and N. Feamster. Challenges and opportunities in Internet data mining. Technical Report CMU–PDL–06–102, CMU Parallel Data Laboratory, Jan. 2006. `http://www.datapository.net/`.

[4] L. Bavoli et al. VisTrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization 2005*, pages 135–142, Oct. 2005.

[5] B. Clark et al. Xen and the art of repeated research. In *Proc. FREENIX Track: 2004 USENIX Annual Tech. Conf.*, pages 135–144, June–July 2004.

[6] Cooperative Association for Internet Data Analysis (CAIDA). DatCat. `http://www.datcat.org/`.

[7] J. Duerig et al. Flexlab: A realistic, controlled, and friendly environment for evaluating networked systems. In *Proc. HotNets V*, pages 103–108, Nov. 2006.

[8] E. Eide, L. Stoller, T. Stack, J. Freire, and J. Lepreau. Integrated scientific workflow management for the Emulab network testbed. In *Proc. USENIX*, pages 363–368, May–June 2006.

[9] Flux Research Group. Emulab tutorial: Link tracing and monitoring. `http://www.emulab.net/tutorial/docwrapper.php3?docname=advanced.html#Tracing`.

[10] Y. E. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. In *Proc. VLDB*, pages 274–285, Sept. 1996.

[11] ISI, University of Southern California. The network simulator - ns-2. `http://www.isi.edu/nsnam/ns/`.

[12] A. J. Lewis. LVM HOWTO. `http://www.tldp.org/HOWTO/LVM-HOWTO`.

[13] B. Ludäscher et al. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice and Experience*, 18(10):1039–1065, Aug. 2006.

[14] J. N. Matthews. The case for repeated research in operating systems. *ACM SIGOPS OSR*, 38(2):5–7, Apr. 2004.

[15] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-aware storage systems. In *Proc. USENIX*, pages 43–56, June 2006.

[16] National Instruments. LabVIEW home page. `http://www.ni.com/labview/`.

[17] T. Oinn et al. Taverna: Lessons in creating a workflow environment for the life sciences. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, Aug. 2006.

[18] V. Paxson. Strategies for sound Internet measurement. In *Proc. 4th ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 263–271, Oct. 2004.

[19] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A blueprint for introducing disruptive technology into the Internet. *ACM SIGCOMM CCR (Proc. HotNets-I)*, 33(1):59–64, Jan. 2003.

[20] R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, 2006. `http://www.r-project.org/`.

[21] R. Ricci et al. The Flexlab approach to realistic evaluation of networked systems. In *Proc. NSDI*, Apr. 2007.

[22] C. Shannon et al. The Internet Measurement Data Catalog. *ACM SIGCOMM CCR*, 35(5):97–100, Oct. 2005.

[23] I. Taylor et al. Distributed computing with Triana on the Grid. *Concurrency and Computation: Practice and Experience*, 17(9):1197–1214, Aug. 2005.

[24] Y. Wang, M. J. Rutherford, A. Carzaniga, and A. L. Wolf. Automating experimentation on distributed testbeds. In *Proc. Conf. on Automated Software Engineering (ASE)*, pages 164–173, Nov. 2005.

[25] K. Webb et al. Implementing the Emulab-PlanetLab portal: Experience and lessons learned. In *Proc. Workshop on Real, Large Dist. Systems (WORLDS)*, Dec. 2004.

[26] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, pages 255–270, Dec. 2002.

[27] J. Yu and R. Buyya. A taxonomy of workflow management systems for Grid computing. Technical Report GRIDS–TR–2005–1, Grid Computing and Distributed Systems Laboratory, Univ. of Melbourne, Mar. 2005.

Black-box and Gray-box Strategies for Virtual Machine Migration

Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif[†]
Univ. of Massachusetts Amherst [†]*Intel, Portland*

Abstract

Virtualization can provide significant benefits in data centers by enabling virtual machine migration to eliminate hotspots. We present Sandpiper, a system that automates the task of monitoring and detecting hotspots, determining a new mapping of physical to virtual resources and initiating the necessary migrations. Sandpiper implements a black-box approach that is fully OS- and application-agnostic and a gray-box approach that exploits OS- and application-level statistics. We implement our techniques in Xen and conduct a detailed evaluation using a mix of CPU, network and memory-intensive applications. Our results show that Sandpiper is able to resolve single server hotspots within 20 seconds and scales well to larger, data center environments. We also show that the gray-box approach can help Sandpiper make more informed decisions, particularly in response to memory pressure.

1 Introduction

Data centers—server farms that run networked applications—have become popular in a variety of domains such as web hosting, enterprise systems, and e-commerce sites. Server resources in a data center are multiplexed across multiple applications—each server runs one or more applications and application components may be distributed across multiple servers. Further, each application sees dynamic workload fluctuations caused by incremental growth, time-of-day effects, and flash crowds [1]. Since applications need to operate above a certain performance level specified in terms of a *service level agreement (SLA)*, effective management of data center resources while meeting SLAs is a complex task.

One possible approach for reducing management complexity is to employ *virtualization*. In this approach, applications run on virtual servers that are constructed using virtual machines, and one or more virtual servers are mapped onto each physical server in the system. Virtualization of data center resources provides numerous benefits. It enables application isolation since malicious or greedy applications can not impact other applications co-located on the same physical server. It enables server consolidation and provides better multiplexing of data center resources across applications. Perhaps the biggest advantage of employing virtualization is the ability to flexibly remap physical resources to virtual servers in order to handle workload dynamics. A workload increase can be handled by increasing the resources allocated to a virtual server, if idle resources are available on the physical server, or by simply migrating the virtual server to a less loaded physical server. Migration is transparent to the applications and all modern virtual machines support this capability [6, 15]. However, detecting workload hotspots and initiating a migration is currently handled manually. Manually-initiated migration lacks the agility to respond to sudden workload changes; it is also error-prone since each reshuffle might require migrations or swaps of multiple virtual servers to rebalance system load. Migration is further complicated by the need to consider multiple resources—CPU, network, and memory—for each application and physical server.

To address this challenge, this paper studies automated black-box and gray-box strategies for virtual machine migration in large data centers. Our techniques automate the tasks of monitoring system resource usage, hotspot detection, determining a new mapping and initiating the necessary migrations. More importantly, our black-box techniques can make these decisions by simply observing each virtual machine from the outside and without any knowledge of the application resident within each VM. We also present a gray-box approach that assumes access to a small amount of OS-level statistics in addition to external observations to better inform the migration algorithm. Since a black-box approach is more general by virtue of being OS and application-agnostic, an important aspect of our research is to understand if a black-box approach alone is sufficient and effective for hotspot detection and mitigation. We have designed and implemented the Sandpiper system to support either black-box, gray-box, or combined techniques. We seek to identify specific limitations of the black-box approach and understand how a gray-box approach can address them.

Sandpiper implements a hotspot detection algorithm that determines *when to migrate* virtual machines, and a hotspot mitigation algorithm that determines *what and where to migrate* and *how much to allocate after the migration*. The hotspot detection component employs a monitoring and profiling engine that gathers usage statistics on various virtual and physical servers and constructs profiles of resource usage. These profiles are used in conjunction with prediction techniques to detect hotspots in the system. Upon detection, Sandpiper's migration manager is invoked for hotspot mitigation. The migration manager employs provisioning techniques to determine the resource needs of overloaded VMs and uses a greedy algorithm to determine a sequence of moves or swaps to migrate overloaded VMs to underloaded servers.

We have implemented our techniques using the Xen virtual machine [3]. We conduct a detailed experimental evaluation on a testbed of two dozen servers using a mix of CPU-, network- and memory-intensive applications. Our results show that Sandpiper can alleviate single server hotspots in less than 20s and more complex multi-server hotspots in a few minutes. Our results show that Sandpiper imposes negligible overheads and that gray-box statistics enable Sandpiper to make better migration decisions when alleviating memory hotspots.

The rest of this paper is structured as follows. Section 2 presents some background, and Sections 3-6 present our design of Sandpiper. Section 7 presents our implementation and evaluation. Finally, Sections 8 and 9 present related work and our conclusions, respectively.

2 Background and System Overview

Existing approaches to dynamic provisioning have either focused on dynamic *replication*, where the number of servers allocated to an application is varied, or dynamic *slicing*, where the fraction of a server allocated to an application is varied; none have considered application *migration* as an option for dynamic provisioning, primarily since migration is not a feasible option in the absence of virtualization. Since migration is transparent to applications executing within virtual machines, our work considers this third approach—resource provisioning via dynamic migrations in virtualized data centers. We present *Sandpiper*[1], a system for automated migration of virtual servers in a data center to meet application SLAs. Sandpiper assumes a large cluster of possibly heterogeneous servers. The hardware configuration of each server—its CPU, network interface, disk and memory characteristics—is assumed to be known to Sandpiper. Each physical server (also referred to as a physical machine or PM) runs a *virtual machine monitor* and one or more virtual machines. Each virtual server runs an application or an application component (the terms virtual servers and virtual machine are used interchangeably). Sandpiper currently uses Xen to implement such an architecture. Each virtual server is assumed to be allocated a certain slice of the physical server resources. In the case of CPU, this is achieved by assigning a weight to the virtual server and the underlying Xen CPU scheduler allocates CPU bandwidth in proportion to the weight. In case of the network interface, Xen is yet to implement a similar fair-share scheduler; a best-effort FIFO scheduler is currently used and Sandpiper is designed to work with this constraint. In case of memory, a slice is assigned by allocating a certain amount of RAM to each resident VM. All storage is assumed to be on a network file system or a storage area network, thereby eliminating the need to move disk state during VM migrations [6].

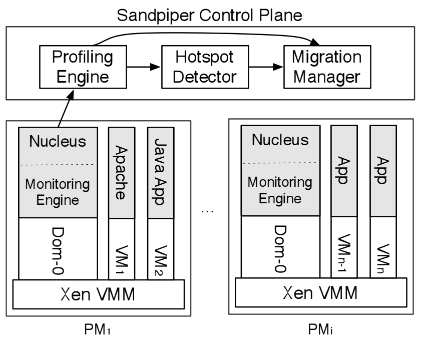


Figure 1: The Sandpiper architecture.

Sandpiper runs a component called the *nucleus* on each physical server; the nucleus runs inside a special virtual server (domain 0 in Xen) and is responsible for gathering resource usage statistics on that server (see Figure 1). It employs a *monitoring engine* that gathers processor, network interface and memory swap statistics for each virtual server. For gray-box approaches, it implements a daemon within each virtual server to gather OS-level statistics and perhaps application logs.

The nuclei periodically relay these statistics to the Sandpiper *control plane*. The control plane runs on a distinguished node and implements much of the intelligence in Sandpiper. It comprises three components: a *profiling engine*, a *hotspot detector* and a *migration manager* (see Figure 1). The profiling engine uses the statistics from the nuclei to construct resource usage profiles for each virtual server and aggregate profiles for each physical server. The hotspot detector continuously monitors

[1]A migratory bird.

these usage profiles to detect hotspots —informally, a hotspot is said to have occurred if the aggregate usage of any resource (processor, network or memory) exceeds a threshold or if SLA violations occur for a "sustained" period. Thus, the hotspot detection component determines *when* to signal the need for migrations and invokes the migration manager upon hotspot detection, which attempts hotspot mitigation via dynamic migrations. It implements algorithms that determine *what* virtual servers to migrate from the overloaded servers, *where* to move them, and *how much* of a resource to allocate the virtual servers once the migration is complete (i.e., determine a new resource allocation to meet the target SLAs). The migration manager assumes that the virtual machine monitor implements a migration mechanism that is transparent to applications and uses this mechanism to automate migration decisions; Sandpiper currently uses Xen's migration mechanisms that were presented in [6].

3 Monitoring and Profiling in Sandpiper

This section discusses online monitoring and profile generation in Sandpiper.

3.1 Unobtrusive Black-box Monitoring

The monitoring engine is responsible for tracking the processor, network and memory usage of each virtual server. It also tracks the total resource usage on each physical server by aggregating the usages of resident VMs. The monitoring engine tracks the usage of each resource over a measurement interval $\mathcal{I}$ and reports these statistics to the control plane at the end of each interval.

In a pure black-box approach, all usages must be inferred solely from external observations and without relying on OS-level support inside the VM. Fortunately, much of the required information can be determined directly from the Xen hypervisor or by monitoring events within domain-0 of Xen. Domain-0 is a distinguished VM in Xen that is responsible for I/O processing; domain-0 can host device drivers and act as a "driver" domain that processes I/O requests from other domains [3, 9]. As a result, it is possible to track network and disk I/O activity of various VMs by observing the driver activity in domain-0 [9]. Similarly, since CPU scheduling is implemented in the Xen hypervisor, the CPU usage of various VMs can be determined by tracking scheduling events in the hypervisor [10]. Thus, black-box monitoring can be implemented in the nucleus by tracking various domain-0 events and without modifying any virtual server. Next, we discuss CPU, network and memory monitoring using this approach.

CPU Monitoring: By instrumenting the Xen hypervisor, it is possible to provide domain-0 with access to CPU scheduling events which indicate when a VM is scheduled and when it relinquishes the CPU. These events are tracked to determine the duration for which each virtual machine is scheduled within each measurement interval $\mathcal{I}$. The Xen 3.0 distribution includes a monitoring application called *XenMon* [10] that tracks the CPU usages of the resident virtual machines using this approach; for simplicity, the monitoring engine employs a modified version of XenMon to gather CPU usages of resident VMs over a configurable measurement interval $\mathcal{I}$.

It is important to realize that these statistics do not capture the CPU overhead incurred for processing disk and network I/O requests; since Xen uses domain-0 to process disk and network I/O requests on behalf of other virtual machines, this processing overhead gets charged to the CPU utilization of domain 0. To properly account for this request processing ovehead, analogous to proper accounting of interrupt processing overhead in OS kernels, we must apportion the CPU utilization of domain-0 to other virtual machines. We assume that the monitoring engine and the nucleus impose negligible overhead and that all of the CPU usage of domain-0 is primarily due to requests processed on behalf of other VMs. Since domain-0 can also track I/O request events based on the number of memory page exchanges between domains, we determine the number of disk and network I/O requests that are processed for each VM. Each VM is then charged a fraction of domain-0's usage based on the proportion of the total I/O requests made by that VM. A more precise approach requiring a modified scheduler was proposed in [9].

Network Monitoring: Domain-0 in Xen implements the network interface driver and all other domains access the driver via clean device abstractions. Xen uses a virtual firewall-router (VFR) interface; each domain attaches one or more virtual interfaces to the VFR [3]. Doing so enables Xen to multiplex all its virtual interfaces onto the underlying physical network interface.

Consequently, the monitoring engine can conveniently monitor each VM's network usage in Domain-0. Since each virtual interface looks like a modern NIC and Xen uses Linux drivers, the monitoring engines can use the Linux `/proc` interface (in particular `/proc/net/dev`) to monitor the number of bytes sent and received on each interface. These statistics are gathered over interval $\mathcal{I}$ and returned to the control plane.

Memory Monitoring: Black-box monitoring of memory is challenging since Xen allocates a user-specified amount of memory to each VM and requires the OS within the VM to manage that memory; as a result, the memory utilization is only known to the OS within each VM. It is possible to instrument Xen to observe memory accesses within each VM through the use of shadow page tables, which is used by Xen's migration mechanism to determine which pages are dirtied during migration. However, trapping each memory access results in a significant application slowdown and is only enabled during migrations[6]. Thus, memory usage statistics are not directly available and must be inferred.

The only behavior that is visible externally is *swap activity*. Since swap partitions reside on a network disk, I/O requests to swap partitions need to be processed by domain-0 and can be tracked. By tracking the reads and writes to each swap partition from domain-0, it is possible to detect memory pressure within each VM. The recently proposed Geiger system has shown that such passive observation of swap activity can be used to infer useful information about the virtual memory subsystem such as working set sizes [11].

Our monitoring engine tracks the number of read and write requests to swap partitions within each measurement interval $\mathcal{I}$ and reports it to the control plane. Since substantial swapping activity is indicative of memory pressure, our current black-box approach is limited to reactive decision making and can not be proactive.

3.2 Gray-box Monitoring

Black-box monitoring is useful in scenarios where it is not feasible to "peek inside" a VM to gather usage statistics. Hosting environments, for instance, run third-party applications, and in some cases, third-party installed OS distributions. Amazon's Elastic Computing Cloud (EC2) service, for instance, provides a "barebone" virtual server where customers can load their own OS images. While OS instrumentation is not feasible in such environments, there are environments such as corporate data centers where both the hardware infrastructure and the applications are owned by the same entity. In such scenarios, it is feasible to gather OS-level statistics as well as application logs, which can potentially enhance the quality of decision making in Sandpiper.

Sandpiper supports gray-box monitoring, when feasible, using a light-weight monitoring daemon that is installed inside each virtual server. In Linux, the monitoring daemon uses the `/proc` interface to gather OS-level statistics of CPU, network, and memory usage. The memory usage monitoring, in particular, enables proactive detection and mitigation of memory hotspots. The monitoring daemon also can process logs of applications such as web and database servers to derive statistics such as request rate, request drops and service times. Direct monitoring of such application-level statistics enables explicit detection of SLA violations, in contrast to the black-box approach that uses resource utilizations as a proxy metric for SLA monitoring.

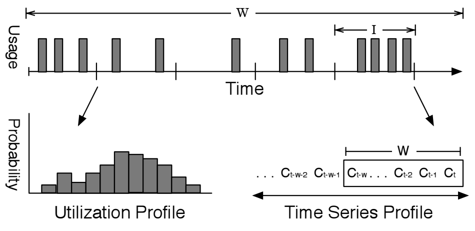

W
I
Usage
Time
Probability
W
. . . C_{t-w-2} C_{t-w-1} C_{t-w} . . . C_{t-2} C_{t-1} C_t
Utilization Profile
Time Series Profile

Figure 2: Profile generation in Sandpiper

3.3 Profile Generation

The profiling engine receives periodic reports of resource usage from each nucleus. It maintains a usage history for each server, which is then used to compute a profile for each virtual and physical server. A profile is a compact description of that server's resouce usage over a sliding time window W. Three black-box profiles are maintained per virtual server: CPU utilization, network bandwidth utilization, and swap rate (i.e., page fault rate). If gray-box monitoring is permitted, four additional profiles are maintained: memory utilization, service time, request drop rate and incoming request rate. Similar profiles are also maintained for each physical server, which indicate the aggregate usage of resident VMs.

Each profile contains a distribution and a time series. The distribution, also referred to as the distribution profile, represents the probability distribution of the resource usage over the window W. To compute a CPU distribution profile, for instance, a histogram of observed usages over all intervals $\mathcal{I}$ contained within the window W is computed; normalizing this histogram yields the desired probability distribution (see Figure 2).

While a distribution profile captures the variations in the resource usage, it does not capture temporal correlations. For instance, a distribution does not indicate whether the resource utilization increased or decreased within the window W. A time-series profile captures these temporal fluctuations and is is simply a list of all reported observations within the window W. For instance, the CPU time-series profile is a list $(C_1, C_2, ..., C_k)$ of the k reported utilizations within the window W. Whereas time-series profiles are used by the hotspot detector to spot increasing utilization trends, distribution profiles are used by the migration manager to estimate peak resource requirements and provision accordingly.

4 Hotspot Detection

The hotspot detection algorithm is responsible for signaling a need for VM migration whenever SLA violations are detected implicitly by the black-box approach or explicitly by the gray-box approach. Hotspot detection is performed on a per-physical server basis in the back-box approach—a hot-spot is flagged if the aggregate CPU or network utilizations on the physical server exceed a threshold or if the total swap activity exceeds a threshold. In contrast, explicit SLA violations must be detected on a per-virtual server basis in the gray-box approach—a hotspot is flagged if the memory utilization of the VM exceeds a threshold or if the response time or the request drop rate exceed the SLA-specified values.

To ensure that a small transient spike does not trigger needless migrations, a hotspot is flagged only if thresholds or SLAs are exceeded for a sustained time. Given a time-series profile, a hotspot is flagged if at least k out the n most recent observations as well as the next predicted value exceed a threshold. With this constraint, we can filter out transient spikes and avoid needless migrations. The values of k and n can be chosen to make hotspot detection aggressive or conservative. For a given n, small values of k cause aggressive hotspot detection, while large values of k imply a need for more sustained threshold violations and thus a more conservative approach. Similarly, larger values of n incorporate a longer history, resulting in a more conservative approach. In the extreme, $n = k = 1$ is the most aggressive approach that flags a hostpot as soon as the threshold is exceeded. Finally, the threshold itself also determines how aggressively hotspots are flagged; lower thresholds imply more aggressive migrations at the expense of lower server utilizations, while higher thresholds imply higher utilizations with the risk of potentially higher SLA violations.

In addition to requiring k out of n violations, we also require that the next predicted value exceed the threshold. The additional requirement ensures that the hotspot is likely to persist in the future based on current observed trends. Also, predictions capture rising trends, while preventing declining ones from triggering a migration.

Sandpiper employs time-series prediction techniques to predict future values [4]. Specifically, Sandpiper relies on the auto-regressive family of predictors, where the n-th order predictor $AR(n)$ uses n prior observations in conjunction with other statistics of the time series to make a prediction. To illustrate the first-order AR(1) predictor, consider a sequence of observations: u_1, u_2, ..., u_k. Given this time series, we wish to predict the demand in the $(k + 1)$th interval. Then the first-order *AR(1) predictor* makes a prediction using the previous value u_k, the mean of the the time series values μ, and the parameter ϕ which captures the variations in the time series [4]. The prediction $\hat{u}_{k+1}$ is given by:

$$\hat{u}_{k+1} = \mu + \phi(u_k - \mu) \tag{1}$$

As new observations arrive from the nuclei, the hot spot detector updates its predictions and performs the above checks to flag new hotspots in the system.

5 Resource Provisioning

A hotspot indicates a resource deficit on the underlying physical server to service the collective workloads of resident VMs. Before the hotspot can be resolved through migrations, Sandpiper must first estimate *how much* additional resources are needed by the overloaded VMs to fulfill their SLAs; these estimates are then used to locate servers that have sufficient idle resources.

5.1 Black-box Provisioning

The provisioning component needs to estimate the *peak* CPU, network and memory requirement of each overloaded VM; doing so ensures that the SLAs are not violated even in the presence of peak workloads.

Estimating peak CPU and network bandwidth needs: Distribution profiles are used to estimate the peak CPU and network bandwidth needs of each VM. The tail of the usage distribution represents the peak usage over the recent past and is used as an estimate of future peak needs. This is achieved by computing a high percentile (e.g., the 95^{th} percentile) of the CPU and network bandwidth distribution as an initial estimate of the peak needs.

Since both the CPU scheduler and the network packet scheduler in Xen are work-conserving, a VM can use more than its fair share, provided that other VMs are not using their full allocations. In case of the CPU, for instance, a VM can use a share that exceeds the share determined by its weight, so long as other VMs are using less than their weighted share. In such instances, the tail of the distribution will exceed the guaranteed share and provide insights into the actual peak needs of the application. Hence, a high percentile of the distribution is a good first approximation of the peak needs.

However, if *all* VMs are using their fair shares, then an overloaded VM will not be allocated a share that exceeds its guaranteed allocation, even though its peak needs are higher than the fair share. In such cases, the observed peak usage (i.e., the tail of the distribution) will equal its fair-share. In this case, the tail of the distribution will *under-estimate* the actual peak need. To correct for this under-estimate, the provisioning component must scale the observed peak to better estimate the actual peak. Thus, whenever the CPU or the network interface on the physical server are close to saturation, the provisioning component first computes a high-percentile of the observed distribution and then adds a constant Δ to scale up this estimate.

Example *Consider two virtual machines that are assigned CPU weights of 1:1 resulting in a fair share of 50% each. Assume that VM$_1$ is overloaded and requires 70% of the CPU to meet its peak needs. If VM$_2$ is underloaded and only using 20% of the CPU, then the work-*

conserving Xen scheduler will allocate 70% to VM$_1$. In this case, the tail of the observed distribution is a good inddicator of VM$_1$'s peak need. In contrast, if VM$_2$ is using its entire fair share of 50%, then VM$_1$ will be allocated exactly its fair share. In this case, the peak observed usage will be 50%, an underestimate of the actual peak need. Since Sandpiper can detect that the CPU is fully utilized, it will estimate the peak to be $50 + \Delta$.

The above example illustrates a fundamental limitation of the black-box approach—it is not possible to estimate the true peak need when the underlying resource is fully utilized. The scale-up factor Δ is simply a guess and might end up over- or under-estimating the true peak.

Estimating peak memory needs: Xen allows a fixed amount of physical memory to be assigned to each resident VM; this allocation represents a hard upper-bound that can not be exceeded regardless of memory demand and regardless of the memory usage in other VMs. Consequently, our techniques for estimating the peak CPU and network usage do not apply to memory. The provisioning component uses observed swap activity to determine if the current memory allocation of the VM should be increased. If swap activity exceeds the threshold indicating memory pressure, then the the current allocation is deemed insufficient and is increased by a constant amount Δ_m. Observe that techniques such as Geiger that attempt to infer working set sizes by observing swap activity [11] can be employed to obtain a better estimate of memory needs; however, our current prototype uses the simpler approach of increasing the allocation by a fixed amount Δ_m whenever memory pressure is observed.

5.2 Gray-box Provisioning

Since the gray-box approach has access to application-level logs, information contained in the logs can be utilized to estimate the peak resource needs of the application. Unlike the black-box approach, the peak needs can be estimated even when the resource is fully utilized.

To estimate peak needs, the peak request arrival rate is first estimated. Since the number of serviced requests as well as the the number of dropped requests are typically logged, the incoming request rate is the summation of these two quantities. Given the distribution profile of the arrival rate, the peak rate is simply a high percentile of the distribution. Let λ_{peak} denote the estimated peak arrival rate for the application.

Estimating peak CPU needs: An application model is necessary to estimate the peak CPU needs. Applications such as web and database servers can be modeled as G/G/1 queuing systems [23]. The behavior of such a G/G/1 queuing system can be captured using the following queuing theory result [13]:

$$\lambda_{cap} \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2 \cdot (d - s)} \right]^{-1} \tag{2}$$

where d is the mean response time of requests, s is the mean service time, and λ_{cap} is the request arrival rate. σ_a^2 and σ_b^2 are the variance of inter-arrival time and the variance of service time, respectively. Note that response time includes the full queueing delay, while service time only reflects the time spent actively processing a request.

While the desired response time d is specified by the SLA, the service time s of requests as well as the variance of inter-arrival and service times σ_a^2 and σ_b^2 can be determined from the server logs. By substituting these values into Equation 2, a lower bound on request rate λ_{cap} that can be serviced by the virtual server is obtained. Thus, λ_{cap} represents the current capacity of the VM.

To service the estimated peak workload λ_{peak}, the current CPU capacity needs to be scaled by the factor $\frac{\lambda_{peak}}{\lambda_{cap}}$. Observe that this factor will be greater than 1 if the peak arrival rate exceeds the currently provisioned capacity. Thus, if the VM is currently assigned a CPU weight w, its allocated share needs to be scaled up by the factor $\frac{\lambda_{peak}}{\lambda_{cap}}$ to service the peak workload.

Estimating peak network needs: The peak network bandwidth usage is simply estimated as the product of the estimated peak arrival rate λ_{peak} and the mean requested file size b; this is the amount of data transferred over the network to service the peak workload. The mean request size can be computed from the server logs.

6 Hotspot Mitigation

Once a hotspot has been detected and new allocations have been determined for overloaded VMs, the migration manager invokes its hotspot mitigation algorithm. This algorithm determines *which* virtual servers to migrate and *where* in order to dissipate the hotspot. Determining a new mapping of VMs to physical servers that avoids threshold violations is NP-hard—the multi-dimensional bin packing problem can be reduced to this problem, where each physical server is a bin with dimensions corresponding to its resource constraints and each VM is an object that needs to be packed with size equal to its resource requirements. Even the problem of determining if a valid packing exists is NP-hard.

Consequently, our hotspot mitigation algorithm resorts to a heuristic to determine which overloaded VMs to migrate and where *such that migration overhead is minimized.* Reducing the migration overhead (i.e., the amount of data transferred) is important, since Xen's live migration mechanism works by iteratively copying the memory image of the VM to the destination while keeping track of which pages are being dirtied and need to be resent. This requires Xen to intercept all memory accesses for the migrating domain, which significantly impacts the performance of the application inside the VM. By reducing the amount of data copied over the network, Sandpiper can minimize the total migration time, and thus, the performance impact on applications. Note that network bandwidth available for application use is also reduced due to the background copying during migrations; however, on a gigabit LAN, this impact is small.

Capturing Multi-dimensional Loads: Once the desired resource allocations have been determined by either our black-box or gray-box approach, the problem of finding servers with sufficient idle resource to house overloaded VMs is identical for both. The migration manager employs a greedy heuristic to determine which VMs need to be migrated. The basic idea is to move load from the most overloaded servers to the least-overloaded servers, while attempting to minimize data copying incurred during migration. Since a VM or a server can be overloaded along one or more of three dimensions–CPU, network and memory–we define a new metric that captures the combined CPU-network-memory load of a virtual and physical server. The *volume* of a physical or virtual server is defined as the product of its CPU, network and memory loads:

$$Vol = \frac{1}{1 - cpu} * \frac{1}{1 - net} * \frac{1}{1 - mem} \qquad (3)$$

where cpu, net and mem are the corresponding utilizations of that resource for the virtual or physical server.[2] The higher the utilization of a resource, the greater the volume; if multiple resources are heavily utilized, the above product results in a correspondingly higher volume. The volume captures the degree of (over)load along multiple dimensions in a unified fashion and can be used by the mitigation algorithms to handle all resource hotspots in an identical manner.

Migration Phase: To determine which VMs to migrate, the algorithm orders physical servers in decreasing order of their volumes. Within each server, VMs are considered in decreasing order of their *volume-to-size ratio (VSR)*; where VSR is defined as *Volume/Size*; size is the memory footprint of the VM. By considering VMs in VSR order, the algorithm attempts to migrate the maximum volume (i.e., load) per unit byte moved, which has been shown to minimize migration overhead [20].

[2]If a resource is fully utilized, its utilization is set to $1 - \epsilon$, rather than one, to avoid infinite volume servers. Also, since the black-box approach is oblivious of the precise memory utilization, the value of mem is set to 0.5 in the absence of swapping and to $1 - \epsilon$ if memory pressure is observed; the precise value of mem is used in the gray-box approach.

The algorithm proceeds by considering the highest VSR virtual machine from the highest volume server and determines if it can be housed on the least volume (least loaded) physical server. The move is feasible only if that server has sufficient idle CPU, network and memory resources to meet the desired resource allocation of the candidate VM as determined by the provisioning component (Section 5). Since we use VSR to represent three resource quantities, the least loaded server may not necessarily "fit" best with a particular VM's needs. If sufficient resources are not available, then the algorithm examines the next least loaded server and so on, until a match is found for the candidate VM. If no physical server can house the highest VSR VM, then the algorithm moves on to the next highest VSR VM and attempts to move it in a similar fashion. The process repeats until the utilizations of all resources on the physical server fall below their thresholds.

The algorithm then considers the next most loaded physical server that is experiencing a hotspot and repeats the process until there are no physcial servers left with a hotspot. The output of this algorithm is a list of overloaded VMs and a new destination server for each; the actual migrations are triggered only after all moves have been determined.

Swap Phase: In cases where there aren't sufficient idle resources on less loaded servers to dissipate a hotspot, the migration algorithm considers VM swaps as an alternative. A swap involves exchanging a high VSR virtual machine from a loaded server with one or more low VSR VMs from an underloaded server. Such a swap reduces the overall utilization of the overloaded server, albeit to a lesser extent than a one-way move of the VM. Our algorithm considers the highest VSR VM on the highest volume server with a hotspot; it then considers the lowest volume server and considers the k lowest VSR VMs such that these VMs collectively free up sufficient resources to house the overloaded VM. The swap is considered feasible if the two physical servers have sufficient resources to house the other server's candidate VM(s) without violating utilization thresholds. If a swap cannot be found, the next least loaded server is considered for a possible swap and so on. The process repeats until sufficient high VSR VMs have been swapped with less loaded VMs so that the hotspot is dissipated. Although multi-way swaps involving more than two servers can also be considered, our algorithm presently does not implement such complex swaps. The actual migrations to perform the swaps are triggered only after a list of all swaps is constructed. Note that a swap may require a third server with "scratch" RAM to temporarily house a VM before it moves to its final destination. An alternative is to (i) suspend one of the VMs on disk, (ii) use the freed up RAM to accommodate the other VM, and (iii)

VM	Peak 1	Peak 2	Peak 3	RAM (MB)	Start PM
1	200	130	130	256	1
2	90	90	90	256	1
3	60	200	60	256	2
4	60	90	90	256	2
5	10	10	130	128	3

Table 1: Workload in requests/second, memory allocations, and initial placement.

resume the first VM on the other server; doing so is not transparent to the temporarily suspended VM.

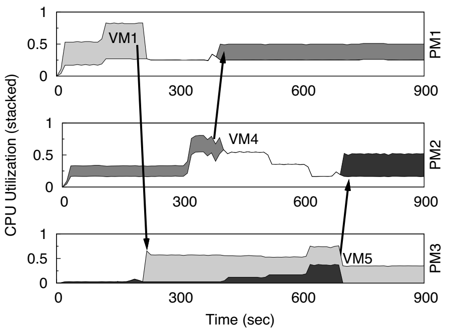


Figure 3: A series of migrations resolve hotspots. Different shades are used for each migrating VM.

7 Implementation and Evaluation

The implementation of Sandpiper is based on Xen. The Sandpiper *control plane* is implemented as a daemon that runs on the control node. It listens for periodic usage reports from the various nuclei, which are used to generate profiles. The profiling engine currently uses a history of the past 200 measurements to generate virtual and physical server profiles. The hotspot detector uses these profiles to detect hotspots; currently a hotspot is triggered when 3 out of 5 past readings and the next predicted value exceeds a threshold. The default threshold is set to 75%. The migration manager implements our provisioning and hotspot mitigation algorithms; it notifies the nuclei of any desired migrations, which then trigger them. In all, the control plane consists of less than 750 lines of python code.

The Sandpiper *nucleus* is a Python application that extends the XenMon CPU monitor to also acquire network and memory statistics for each VM. The monitoring engine in the nucleus collects and reports measurements once every 10 seconds—the default measurement interval. The nucleus uses Xen's Python management API to trigger migrations and adjust resource allocations as directed by the control plane. While black-box monitoring only requires access to domain-0 events, gray-box monitoring employs two additional components: a Linux OS daemon and an Apache module.

The gray-box linux daemon runs on each VM that permits gray-box monitoring. It currently gathers memory statistics via the `/proc` interface—the memory utilization, the number of free pages and swap usage are reported to the monitoring engine in each interval. The gray-box apache module comprises of a real-time log analyzer and a dispatcher. The log-analyzer processes log-entries as they are written to compute statistics such as the service time, request rate, request drop rate, interarrival times, and request/file sizes. The dispatcher is implemented as a kernel module based on Linux IP Virtual server (IPVS) ver 1.2.1; the goal of the kernel module is to accurately estimate the request arrival rate during overload periods, when a high fraction of requests may be dropped. Since requests can be dropped at the TCP layer as well as at the HTTP layer during overloads, the use of a transport-level dispatcher such as IPVS is necessary for accurately estimating the drop (and hence arrival) rates. Ordinarily, the kernel dispatcher simply forwards incoming requests to Apache for processing. In all, the nucleus comprises 650 lines of Python code.

Our evaluation of Sandpiper is based on a prototype data center consisting of twenty 2.4Ghz Pentium-4 servers connected over gigabit ethernet. All servers run Linux 2.6.16 and Xen 3.0.2-3 and are equipped with at least 1GB of RAM. A cluster of Pentium-3 Linux servers is used to generate workloads for our experiments. One node in the cluster is designated to run the Sandpiper control plane, while the rest host one or more VMs, all of which run the Sandpiper nucleus in domain- 0. In the following experiments, our VMs run Apache 2.0.54, PHP 4.3.10, and MySQL 4.0.24.

7.1 Migration Effectiveness

Our first experiment demonstrates the effectiveness of migrations and swaps by exercising Sandpiper's hotspot detection and migration algorithms; we subject a set of black-box servers to a series of workloads that repeatedly place the system in overload. Our experiment uses three physical servers and five VMs with memory allocations as shown in Table 1. All VMs run Apache serving dynamic PHP web pages. The PHP scripts are designed to be CPU intensive so that a low client request rate places a large CPU load on a server without significant network or memory utilization. We use *httperf* to inject a workload that goes through three phases, each of which causes a hotspot on a different physical machine. The peak request rates for each phase are shown in Table 1.

Figure 3 presents a time series of the load placed on each VM along with the triggered migrations. In the first phase, a large load is placed on VM_1, causing the CPU utilization on PM_1 to exceed the CPU threshold. The system detects a hotspot at t=166 seconds. The migration manager examines candidates for migration in VSR order. VM_1 has the highest VSR, so it is selected as a candidate. Since PM_3 has sufficient spare capacity to house VM_1, it is migrated there, thereby eliminating the hotspot. This represents the ideal case for our algorithm: if possible, we try to migrate the most loaded VM from an overloaded PM to one with sufficient spare capacity.

In the second phase, PM_2 becomes overloaded due to increasing load on VM_3. However, the migration manager is unable to migrate this VM because there is insufficient capacity on the other PMs. As a result, at t=362 seconds, the VM on PM_2 with the second highest VSR VM_4, is migrated to PM_1 that now has spare capacity. This demonstrates a more typical case where none of the underloaded PMs have sufficient spare capacity to run the overloaded PM's highest VSR VM, so instead we migrate less overloaded VMs that can fit elsewhere.

In the final phase, PM_3 becomes overloaded when both of its VMs receive *identical* large loads. Unlike the previous two cases where candidate VMs had identical memory footprints, VM_5 has half as much RAM as VM_1, so it is chosen for migration. By selecting the VM with a lower footprint, Sandpiper maximizes the reduction in load per megabyte of data transfered.

Result: To eliminate hotspots while minimzing the overhead of migration, our placement algorithm tries to move the highest VSR VM to the least loaded PM. This maximizes the amount of load displaced from the hotspot per megabyte of data transferred.

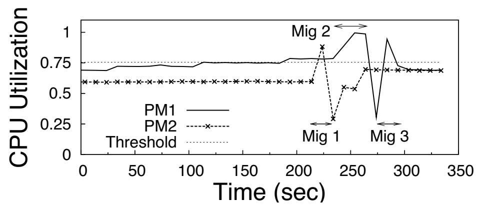
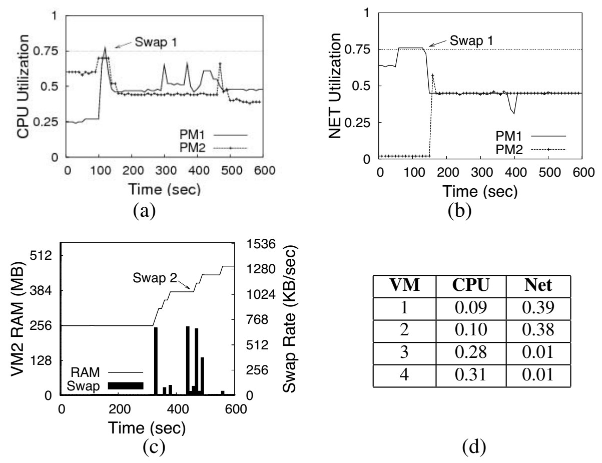


Figure 4: After a hotspot is detected on PM_1, a swap occurs via the scratch PM. Initially, VM_1 and VM_2 are on PM_1. VM_2 is swapped with VM_4 from PM_2 to resolve overload.



VM	CPU	Net
1	0.09	0.39
2	0.10	0.38
3	0.28	0.01
4	0.31	0.01

Figure 5: Swaps and migrations to handle network- and memory-intensive loads. Initially, VM_1 and VM_2 are on PM_1, the rest on PM_2. After two swaps, PM_1 hosts VM_1 and VM_4.

7.2 Virtual Machine Swaps

Next we demonstrate how VM swaps can mitigate hotspots. The first two VMs in our setup are allocated 384 MB of RAM on PM_1; VM_3 and VM_4 are assigned 256 MB each on PM_2. The load on VM_1 steadily increases during the experiment, while the others are constant. As before, clients use httperf to request dynamic PHP pages.

Figure 4 shows that a hotspot is detected on PM_1 due to the increasing load on VM_1. However, there is insufficient spare capacity on PM_2 to support a migrated VM. The only viable solution is to swap VM_2 with VM_4. To facilitate such swaps, Sandpiper uses spare RAM on the control node as scratch space.

By utilizing this scratch space, Sandpiper never requires either physical server to simultaneously run both VMs. It does require us to perform three migrations instead of two; however, Sandpiper reduces the migration cost by always migrating the smaller footprint VM via the scratch server. As shown in Figure 4, the load on PM_2 drops at t=219 due to the migration of VM_4 to scratch. VM_2 is then migrated directly from PM_1 to PM_2 at t=233, followed by a migration of VM_4 from scratch to PM_1, which takes an additional 10s.

The figure also depicts the CPU overhead of a migration—as indicated by a rise in CPU utilization of the initiating server whenever a migration begins. This suggests using lower CPU hotspot thresholds to safely absorb the additional overheads caused by a migration.

Result: Swaps incur more overhead, but increase the chances of mitigating hotspots in clusters with high average utilization.

7.3 Mixed Resource Workloads

Sandpiper can consolidate applications that stress different resources to improve the overall multiplexing of server resources. Our setup comprises two servers with two VMs each. Both VMs on the first server are network-intensive, involving large file transfers, while those on the second server are CPU-intensive running Apache with dynamic PHP scripts. All VMs are initially allocated 256 MB of memory. VM_2 additionally runs a main-memory database that stores its tables in memory, causing its memory usage to grow over time.

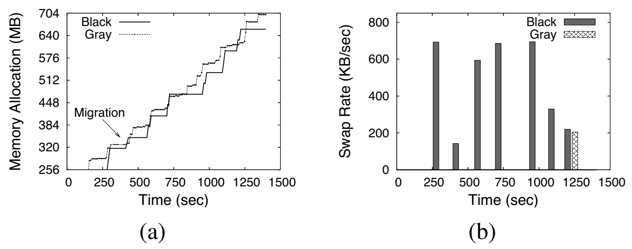


Figure 6: The black-box system lags behind the gray-box system in allocating memory. With its detailed memory statistics, the gray-box approach is proactive and increases memory without waiting until swapping occurs.

Figures 5(a) and (b) show the resource utilization of each PM over time. Since PM_1 has a network hotspot and PM_2 has a CPU hotspot, Sandpiper swaps a network-intensive VM for a CPU-intensive VM at t=130. This results in a lower CPU and network utilization on both servers. Figure 5(d) shows the initial utilizations of each VM; after the swap, the aggregate CPU and network utilizations on both servers falls below 50%.

In the latter half, memory pressure increases on VM_2 due to its main-memory database application. As shown in 5(c), Sandpiper responds by increasing the RAM allocation in steps of 32MB every time swapping is observed; when no additional RAM is available, the VM is swapped to the second physical server at t=430. This is feasible because two cpu-intensive jobs are swapped, leaving CPU and network utilization balanced, and the second physical server has more RAM than the first. Memory allocations are reactive since only black-box stats are available. Next we demonstrate how a gray-box approach can proactively respond to memory pressure.

Result: Sandpiper can respond to network, CPU, or memory hotspots and can collocate VMs that stress different resources to improve overall system utilization.

7.4 Gray v. Black: Memory Allocation

We compare the effectiveness of the black- and gray-box approaches in mitigating memory hotspots using the SPECjbb 2005 benchmark. SPECjbb emulates a three-tier web application based on J2EE servers. We use SPECjbb to apply an increasingly intense workload to a single VM. The workload increases every two minutes, causing a significant increase in memory usage.

The VM is initially assigned 256MB of RAM, and resides on a physical machine with 384 MB total RAM. We also run a second, idle physical server which has 1GB RAM. We run the experiment with two separate pairs of servers, Black and Gray, that correspond to the black- and gray-box approaches, respectively. The Gray system is configured to signal a hotspot whever the amount of

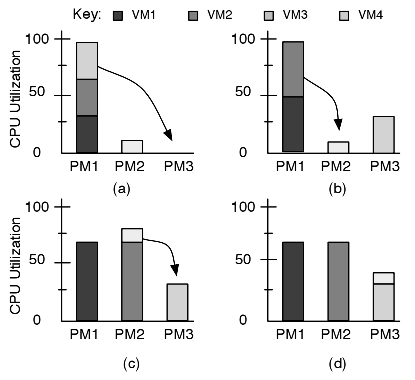


Figure 7: The black-box system incorrectly guesses resource requirements since CPU usage is saturated, resulting in an increased resolution time. The gray-box system infers usage requirements and transitions directly from a) to d).

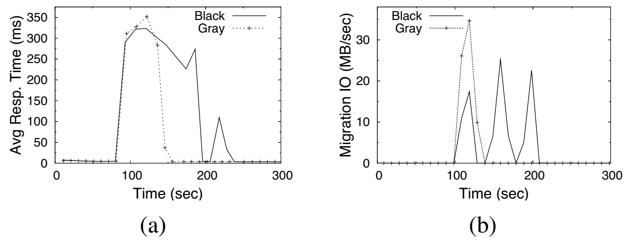


Figure 8: The gray-box system balances the system more quickly due to more informed decision making. The black-box system must perform migrations sequentially and incurs an additional migration.

free RAM in the virtual machine falls below 32MB.

Fig. 6(a) plots the memory allocation of the VM over time. Both systems gradually increase the VM's memory until all unused RAM is exhausted. Since Black can only respond to swapping, it lags in responsiveness.

At t=380 seconds, Gray determines that there is insufficient RAM for the VM and migrates it to the second PM with 1GB RAM; Black initiates the same migration shortly afterward. Both continue to increase the VM's memory as the load rises. Throughout the experiment, Black writes a total of 32MB to swap, while Gray only writes 2MB. Note that a lower memory hotspot threshold in Gray can prevent swapping altogether, while Black can not eliminate swapping due to its reactive nature.

Result: A key weakness of the black-box approach is its inability to infer memory usage. Using this information, the gray-box system can reduce or eliminate swapping without significant overprovisioning of memory.

7.5 Gray v. Black: Apache Performance

Next, we compare the effectiveness of black- and gray-box approaches in provisioning resources for overloaded

VMs. Recall from Section 5 that when resources are fully utilized, they hamper the black-box approach from accurately determining the needs of overloaded VMs. Our experiment demonstrates how a black-box approach may incur extra migrations to mitigate a hotspot, whereas a gray-box approach can use application-level knowledge for faster hotspot mitigation.

Our experiment employs three physical servers and four VMs. Initially, VM_1 through VM_3 reside on PM_1, VM_4 resides on PM_2, and PM_3 is idle. We use httperf to generate requests for CPU intensive PHP scripts on all VMs. At t=80s, we rapidly increase the request rates on VM_1 and VM_2 so that actual CPU requirement for *each* virtual machine reaches 70%. The request rates for VM_3 and VM_4 remain constant, requiring 33% and 7% CPU respectively. This creates an extreme hotspot on PM_1. We configure Sandpiper with an aggressive 6 second measurement interval so that it can respond quickly to the increase in workload.

Without accurate estimates of each virtual machine's resource requirements, the black-box system falters in its decision making as indicated in Figure 7. Since the CPU on PM_1 is saturated, each virtual machine receives an equal portion of processing time and appears equivalent to Sandpiper. Sandpiper must select a VM at random, and in the worst case, tries to eliminate the hotspot by migrating VM_3 to PM_3. Since VM_1 and VM_2 continue to reside on PM_1, the hotspot persists even after the first migration. Next, the black-box approach assumes that VM_2 requires only 50% of the CPU and migrates it to PM_2. Unfortunately, this results in PM_2 becoming overloaded, so a final migration must be performed to move VM_4 to PM_3.

We repeat this scenario with the Apache gray-box module running inside of each virtual machine. Since the gray-box monitor can precisely measure the incoming request rates, Sandpiper can accurately estimate the CPU needs of VM_1 and VM_2. By using this information, Sandpiper is able to efficiently respond to the hotspot by immediately migrating VM_3 to PM_2 and VM_2 to PM_3. Figure 8 depicts the improved performance of the gray-box approach. Note that since Sandpiper requires the hotspot to persist for k out of n intervals before it acts, it is not until $t = 98$s that either system considers itself overloaded. Once a hotspot is flagged, the gray-box approach can mitigate it within 40 seconds with just two migrations, while the black-box approach requires 110 seconds and three migrations to do so. Although response time increases equally under both systems, the gray-box approach is able to reduce response times to an acceptable level 61% faster than the black-box system, producing a corresponding reduction in SLA violations.

Result: Application-level statistics enable the gray-box approach to better infer resource needs and improves the quality of migration decisions, especially in scenarios where resource demands exceed server capacity.

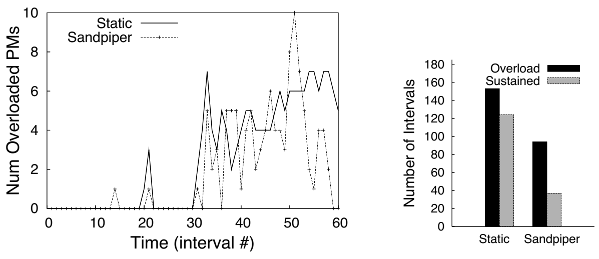


Figure 9: Sandpiper eliminates all hotspots and reduces the number of intervals experiencing sustained overload by 61% .

7.6 Prototype Data Center Evaluation

Next we conduct an experiment to demonstrate how Sandpiper performs under realistic data center conditions. We deployed a prototype data center on a cluster of 16 servers that run a total of 35 VMs. An additional node runs the control plane and one node is reserved as a scratch node for swaps. The virtual machines run a mix of data center applications ranging from Apache and streaming servers to LAMP servers running Apache, PHP, and MySQL within a single VM. We run RUBiS on our LAMP servers—RUBiS is an open-source multi-tier web application that implements an eBay-like auction web site and includes a workload generator that emulates users browsing and bidding on items.

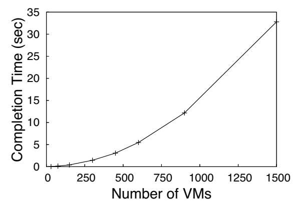
Of the 35 deployed VMs, 5 run the RUBiS application, 5 run streaming servers, 5 run Apache serving CPU-intensive PHP scripts, 2 run main memory database applications, and the remaining 15 serve a mix of PHP scripts and large HTML files. We use the provided workload generators for the RUBiS applications and use httperf to generate requests for the other servers.

To demonstrate Sandpiper's ability to handle complex hotspot scenarios, we orchestrate a workload that causes multiple network and CPU hotspots on several servers. Our workloads causes six physical servers running a total of 14 VMs to be overloaded—four servers see a CPU hotspot and two see a network hotspot. Of the remaining PMs, 4 are moderately loaded (greater than 45% utilization for at least one resource) and 6 have lighter loads of between 25 and 40% utilization. We compare Sandpiper to a statically allocated system with no migrations.

Figure 9 demonstrates that Sandpiper eliminates hotspots on all six servers by interval 60. These hotspots persist in the static system until the workload changes or a system administrator triggers manual migrations. Due to Xen's migration overhead, there are brief periods where Sandpiper causes more physical servers to be

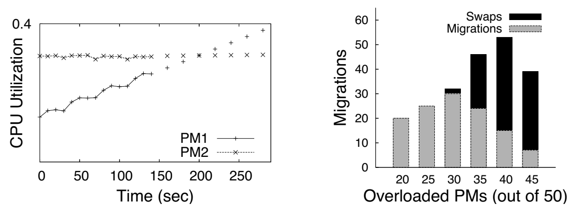
Log Type	Bytes
Black-box	178
Gray: Mem	60
Gray: Apache	50
Total	288

(a)



(b)

Figure 10: Sandpiper overhead and scalability



(a) (b)

Figure 11: (a) Using time series predictions (the dotted lines) allows Sandpiper to better select migration destinations, improving stability. (b) Higher levels of overload requires more migrations until there is no feasible solution.

overloaded than in the static case. Despite this artifact, even during periods where migrations are in progress, Sandpiper reduces the number of intervals spent in sustained overload by 61%. In all, Sandpiper performs seven migrations and two swaps to eliminate all hotspots over a period of 237 seconds after hotspot detection.

Result: Sandpiper is capable of detecting and eliminating simultaneous hotspots along multiple resource dimensions. Despite Xen's migration overhead, the number of servers experiencing overload is decreased even while migrations are in progress.

7.7 System Overhead and Scalability

Sandpiper's CPU and network overhead is dependent on the number of PMs and VMs in the data center. With only black-box VMs, the type of application running in the VM has no effect on Sandpiper's overhead. If gray-box modules are in use, the overhead may vary depending on the size of application-level statistics gathered.

Nucleus Overheads: Sandpiper's usage reports are sent from each nucleus to the Control Plane every measurement interval (10 seconds by default). The table in Figure 10(a) gives a breakdown of overhead for each report type. Since each report uses only 288 bytes per VM, the resulting overhead on a gigabit LAN is negligible. To evaluate the CPU overhead, we compare the performance of a CPU benchmark with and without our resource monitors running. Even on a single physical server running 24 concurrent VMs, our monitoring overheads reduce the CPU benchmark by approximately one percent. This is comparable to the overheads reported by XenMon, which much of our code is based on [10]. We also compare the performance of an Apache server with and without our monitoring software and find no significant difference in response time or attainable throughput between the two scenarios.

Control Plane Scalability: The main source of computational complexity in the control plane is the computation of a new mapping of virtual machines to physical servers after detecting a hotspot. Although the problem is NP-hard, we only require an approximate solution, and our heuristics make the problem tractable for reasonable system sizes. For data centers with up to 500 virtual servers, the algorithm completes in less than five seconds as shown in Figure 10(b). For very large data centers with thousands of virtual machines, the computation could be split up accross multiple nodes, or the center's servers can be broken up into pools, each controlled independently by its own control plane.

Result: Our measurement system has insignificant CPU and I/O requirements and has a negligible impact on performance. The computation time of our placement algorithm is the main limit on scalability, but the computations can be distributed if necessary.

7.8 Stability During Overloads

This section demonstrates how Sandpiper ensures stable system behavior by avoiding "thrashing" migrations. First, Sandpiper avoids migrations to physical machines with rising loads, since this can trigger additional migrations if the load rises beyond the threshold; time-series predictions are used to determine future load trends when selecting a physical server. Thus, Figure 11(a) shows that when a migration decision is required at t=140 sec, Sandpiper will prefer PM_2 over PM_1 as a target. Even though PM_2 has a *higher* current load, the 120 second prediction window indicates a rising load on PM_1.

Next, we demonstrate Sandpiper's behavior in the presence of increasing number of hotspots. We simulate a data center with fifty physical servers, each with three virtual servers. We increase the number of simultaneous hotspots from 20 to 45; the mean utilizations are set to 85% and 45% for servers with and without hotspots. Figure 11(b) depicts the mean number of migrations performed to resolve these hotspots over multiple runs. If fewer than half of the servers are overloaded, then all hotspots can typically be resolved with one migration per overloaded server. After this threshold, swaps are required and it is increasingly difficult to fully resolve overload until it becomes infeasible. With 35 overloaded servers, Sandpiper was able to eliminate all hotspots 73% of the time (over multiple runs); with 40 overloaded servers, a complete solution was found only 3% of the time. In the extreme case, Sandpiper is still able to resolve 22 of the 45 hotspots before giving up. In all cases, Sandpiper first finds a solution before initiating migrations or swaps; when no feasible solutions are found, Sandpiper either implements a partial solution or gives up entirely rather than attempting wasteful migrations. This bounds the number of migrations which will ever be performed and explains the decrease in migrations beyond 40 overloaded servers, where there is no feasible solution.

7.9 Tuning Sandpiper

Sandpiper has several parameters which the system administrator can tune to make hotspot detection and mitigation more or less aggressive. Our experiments suggest the following rules of thumb:

Setting Thresholds: If overload thresholds are set too high, then the additional overhead during migration can cause additional SLA violations. Our experiments show that the average throughput of a CPU-intensive Apache server can drop by more than 50% during a migration. We suggest a CPU threshold of 75% to absorb the CPU overhead of migration while maximizing server utilization. We also suggest a 75% threshold for network utilization based on experiments in [6] which indicate that the network throughput of a highly loaded server can drop by about 20% during portions of a migration (due to network copying overheads).

Sustained Overload Requirement: Our experiments (not reported here) reveal that Sandpiper is not sensitive to a particular choice of the measurement interval $\mathcal{I}$ so long as it is between a few seconds and a few tens of seconds. For a measurement interval of 10s, we suggest $k = 3$ and $n = 5$ for the "k out of n" check; this corresponds to requiring the time period of about 3 migrations to exceed the resource threshold before we initiate a migration. The Δ paramter is used in the black-box system to increase resource allocations when utilization is saturated. This should be set equal to the maximum increase in resource requirements that a service is likely to see during a measurement interval and may vary based on workload; we use 10% in our experiments. Using more advanced time series forecasting techniques would allow Sandpiper to dynamically determine Δ.

8 Related Work

Our work draws upon recent advances in virtual machines and dynamic provisioning in data centers to address a question of increasing research and commercial interest: can virtual machine migration enable robust and highly responsive provisioning in data centers? The Xen migration work [6] alludes to this motivation. What is missing is a convincing validation and algorithms to effect migration, which is the focus of this paper.

The idea of process migration was first investigated in the 80's [22]. Support for migrating groups of processes across OSes was presented in [16], but applications had to be suspended and it did not address the problem of maintaining open network connections. Virtualization support for commodity operating systems in [7] led towards techniques for virtual machine migration over long time spans, suitable for WAN migration [19]. More recently, Xen [6] and VMWare [15] have implemented "live" migration of VMs that involve extremely short downtimes ranging from tens of milliseconds to a second. VM migration has been used for dynamic resource allocation in Grid environments [18, 21, 8]. A system employing automated VM migrations for scientific nano-technology workloads on federated grid environments was investigated in [18]. The Shirako system provides infrastructure for leasing resources within a federated cluster environment and was extended to use virtual machines for more flexible resource allocation in [8]. Shirako uses migrations to enable dynamic placement decisions in response to resource broker and cluster provider policies. In contrast, we focus on data center environments with stringent SLA requirements that necessitate highly responsive migration algorithms for online load balancing. VMware's Distributed Resource Scheduler [24] uses migration to perform automated load balancing in response to CPU and memory pressure. DRS uses a userspace application to monitor memory usage similar to Sandpiper's gray box monitor, but unlike Sandpiper, it cannot utilize application logs to respond directly to potential SLA violations or to improve placement decisions.

Dedicated hosting is a category of dynamic provisioning in which each physical machine runs at most one application and workload increases are handled by spawning a new replica of the application on idle servers. Physical server granularity provisioning has been investigated in [1, 17]. Techniques for modeling and provisioning multi-tier Web services by allocating physical machines to each tier are presented in [23]. Although dedicated hosting provides complete isolation, the cost is reduced responsiveness - without virtualization, moving from one physical machine to another takes on the order of several minutes [23] making it unsuitable for handling flash crowds. Our current implementation does not replicate virtual machines, implicitly assuming that PMs are sufficiently provisioned.

Shared hosting is the second variety of dynamic provisioning, and allows a single physical machine to be shared across multiple services. Various economic and resource models to allocate shared resources have been presented in [5]. Mechanisms to partition and share resources across services include [2, 5]. A dynamic provisioning algorithm to allocate CPU shares to VMs on a single physical machine (as opposed to a cluster) was presented and evaluated through simulations in [14]. In comparison to the above systems, our work assumes a shared hosting platform and uses VMs to partition CPU, memory, and network resources, but additionally leverages VM migration to meet SLA objectives.

Estimating the resources needed to meet an application's SLA requires a model that inspects the request arrival rates for the application and infers its CPU, memory, and network bandwidth needs. Developing such models is not the focus of this work and has been addressed by several previous efforts such as [12, 1].

9 Conclusions And Future Work

This paper argued that virtualization provides significant benefits in data centers by enabling virtual machine migration to eliminate hotspots. We presented Sandpiper, a system that automates the task of monitoring and detecting hotspots, determining a new mapping of physical to virtual resources and initiating the necessary migrations in a virtualized data center. We discussed a black-box strategy that is fully OS- and application-agnostic as well as a gray-box approach that can exploit OS- and application-level statistics. An evaluation of our Xen-based prototype showed that VM migration is a viable technique for rapid hotspot elimination in data center environments. Using solely black-box methods, Sandpiper is capable of eliminating simultaneous hotspots involving multiple resources. We found that utilizing gray-box information can improve the responsiveness of our system, particularly by allowing for proactive memory allocations and better inferences about resource requirements. In future work, we plan to extend Sandpiper to support replicated services by automatically determining whether to migrate a VM or to spawn a replica in order to garner more resources.

Acknowledgements: We would like to thank our anonymous reviewers and our shepherd, Anees Shaikh, for their helpful comments. This research was supported by NSF grants EEC-0313747, CNS-0323597, CNS-0325868 and a gift from Intel.

References

[1] K. Appleby, S. Fakhouri, L. Fong, M. Goldszmidt, S. Krishnakumar, D. Pazel, J. Pershing, and B. Rochwerger. Oceano - sla-based management of a computing utility. In *Proc. IFIP/IEEE Symposium on Integrated Management*, May 2001.

[2] M. Aron, P. Druschel, and W. Zwaenepoel. Cluster reserves: A mechanism for resource management in cluster-based network servers. In *Proc. ACM SIGMETRICS '00*.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proc. SOSP'03*, pages 164–177, October 2003.

[4] G. P. Box, G. M. Jenkins, and G. C. Reinsel. *Time Series Analysis Forecasting and Control Third Edition*. Prentice Hall, 1994.

[5] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proc. SOSP '01*.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. NSDI '05*, May 2005.

[7] K. Govil, D. Teodosiu, Y. Huang, and M. Rosenblum. Cellular disco: Resource management using virtual clusters on shared-memory multiprocessors. In *Proc. SOSP'99*, pages 154–169, December 1999.

[8] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual Machine Hosting for Networked Clusters: Building the Foundations for Autonomic Orchestration. In *Proc. VTDC '06*.

[9] D. Gupta, L. Cherkasova, R. Gardner, and A. Vahdat. Enforcing Performance Isolation Across Virtual Machines in Xen. In *Proc. Middleware '06*, October, 2006.

[10] D. Gupta, R. Gardner, and L. Cherkasova. Xenmon: Qos monitoring and performance profiling tool. Technical Report HPL-2005-187, HP Labs, 2005.

[11] S. Jones, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Geiger: Monitoring the buffer cache in a virtual machine environment. In *Proc. ASPLOS'06*, pages 13–23, October 2006.

[12] A. Kamra, V. Misra, and E. Nahum. Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites. In *Proc. IWQoS '04*, June 2004.

[13] L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. John Wiley and Sons, Inc., 1976.

[14] D. Menasce and M. Bennani. Autonomic Virtualized Environments. In *IEEE ICAS 06*.

[15] M. Nelson, B. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proc. USENIX* 2005.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments, In *Proc. OSDI* 2002.

[17] S. Ranjan, J. Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. In *Proc. IWQoS 2002*.

[18] P. Ruth, J. Rhee, D. Xu, R. Kennell, and S. Goasguen. Autonomic Live Adaptation of Virtual Computational Environments in a Multi-Domain Infrastructure. In *Proc. IEEE ICAC '06*.

[19] C. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Proc. OSDI '02*.

[20] V. Sundaram, T. Wood, and P. Shenoy. Efficient Data Migration in Self-managing Storage Systems. In *Proc. ICAC '06*.

[21] A. Sundararaj, A. Gupta, and P. Dinda. Increasing Application Performance in Virtual Environments through Run-time Inference and Adaptation. In *Proc. HPDC '05*.

[22] M. M. Theimer, K. A. L., and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proc. SOSP* December 1985.

[23] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning for multi-tier internet applications. In *Proc. ICAC '05*, June 2005.

[24] VMware Dynamic Resource Scheduler. http://www.vmware.com/products/vi/vc/drs.html.

Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code

Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat
University of California, San Diego
{*ckillian, jwanderson, jhala, vahdat*}@*cs.ucsd.edu*

Abstract

Modern software model checkers find *safety* violations: breaches where the system enters some bad state. However, we argue that checking *liveness* properties offers both a richer and more natural way to search for errors, particularly in complex concurrent and distributed systems. Liveness properties specify desirable system behaviors which must be satisfied *eventually*, but are not *always* satisfied, perhaps as a result of failure or during system initialization.

Existing software model checkers cannot verify liveness because doing so requires finding an infinite execution that does not satisfy a liveness property. We present heuristics to find a large class of liveness violations and the *critical transition* of the execution. The critical transition is the step in an execution that moves the system from a state that does not currently satisfy some liveness property—but where recovery is possible in the future—to a dead state that can never achieve the liveness property. Our software model checker, MACEMC, isolates complex liveness errors in our implementations of PASTRY, CHORD, a reliable transport protocol, and an overlay tree.

1 Introduction

Hard-to-find, non-reproducible bugs have long been the bane of systems programmers. Such errors prove especially challenging in unreliable distributed environments with failures and asynchronous communication. For example, we have run our MACE implementation of the PASTRY [28] overlay on the Internet and emulated environments for three years with occasional unexplained erroneous behavior: some nodes are unable to rejoin the overlay after restarting. Unable to recreate the behavior, we never succeeded in tracking down the cause of the error.

Motivated by this and similarly subtle bugs, we turned to model checking to assist us in building robust distributed systems. Unfortunately, existing model checkers able to run on systems implementations (rather than specifications) can only find *safety* violations—counterexamples of a specified condition that should always be true. Simple examples of safety properties are `a    r` statements and unhandled program exceptions. For our target systems however, specifying global *liveness* properties—conditions that should always *eventually* be true—proved to be more desirable. In the above example, we wished to verify that eventually all PASTRY nodes would form a ring. Somewhat paradoxically, specifying the appropriate safety property requires knowledge of the nature of the bug, whereas specifying the appropriate liveness property only requires knowledge of desirable high-level system properties. It is acceptable for a node to be unable to join a ring temporarily, but in our case, the bug made it impossible for a node to ever join the ring, thus violating liveness.

Existing software model checkers focus on safety properties because verifying liveness poses a far greater challenge: the model checker cannot know *when* the properties should be satisfied. Identifying a liveness violation requires finding an *infinite* execution that will not ever satisfy the liveness property, making it impractical to find such violating infinite executions in real implementations. Thus, we set out to develop practical heuristics that enable software model checkers to determine whether a system satisfies a set of liveness properties.

We present MACEMC, the first software model checker that helps programmers find liveness violations in complex systems implementations. We built our solution upon three key insights:

Life: To find subtle, complicated bugs in distributed systems, we should search for liveness violations in addition to safety violations. Liveness properties free us from only specifying what ought not happen—that is, error conditions and invariants, which may be hopelessly complicated or simply unknown—and instead let us specify what ought to happen.

Death: Instead of searching for general liveness violations, which require finding violating infinite executions, we focus on a large subset: those that enter *dead* states from which liveness can never be achieved regardless of any subsequent actions. We thereby reduce the problem of determining liveness to searching for violations of previously unknown safety properties. We present a novel heuristic to identify dead states and locate executions leading to them by combining exhaustive search with long random executions.

Critical Transition: To understand and fix a liveness error, the developer must painstakingly analyze the tens of thousands of steps of the non-live execution to find where and how the system became dead. We show how to extend our random execution technique to automatically search for the *critical transition*, the step that irrecoverably cuts off all possibility of ever reaching a live state in the future.

To further help the programmer understand the cause of an error, we developed Mdb, an interactive debugger providing forward and backward stepping through global events, per-node state inspection, and event graph visualization. In our experience, Mdb, together with the critical transition automatically found by MaceMC, reduced the typical human time required to find and fix liveness violations from a few hours to less than 20 minutes.

Using MaceMC and Mdb, we found our Pastry bug: under certain circumstances, a node attempting to rejoin a Pastry ring using the same identifier was unable to join because its join messages were forwarded to unjoined nodes. This error was both sufficiently obscure and difficult to fix that we decided to check how FreePastry [1], the reference implementation, dealt with this problem. The following log entry in a recent version of the code (1.4.3) suggests that FreePastry likely observed a similar problem: "Dropped JoinRequest on rapid rejoin problem – There was a problem with nodes not being able to quickly rejoin if they used the same NodeId. Didn't find the cause of this bug, but can no longer reproduce."

We have found 52 bugs using MaceMC thus far across a variety of complex systems. While our experience is restricted to MaceMC, we believe our random execution algorithms for finding liveness violations and the critical transition generalize to any state-exploration model checker capable of replaying executions. It should therefore be possible to use this technique with systems prepared for other model checkers by defining liveness properties for those systems. Although our approach to finding liveness violations is necessarily a heuristic—a proof of a liveness violation requires finding an infinite execution that never satisfies liveness—we have not had any false positives among the set of identified violations to date.

2 System Model

Software model checkers find errors by exploring the space of possible executions for systems implementations. We establish the MaceMC system model with our simplified definitions of programs and properties (see [19] for the classical definitions). We then discuss the relationship between liveness and safety properties.

Distributed Systems as Programs We model-check distributed systems by composing every node and a simulated network environment in a single program (cf. §4.1 for the details of preparing unmodified systems for model checking). A program *state* is an assignment of values to variables. A *transition* maps an input state to an output state. A *program* comprises a set of variables, a set of initial states, and a set of transitions. A *program execution* is an infinite sequence of states, beginning in an initial program state, with every subsequent state resulting from the application of some transition (an atomic set of machine instructions) to its predecessor. Intuitively, the set of variables corresponds to those of every node together with the distributed environment, such as the messages in the network. Thus, a state encodes a snapshot of the entire distributed system at a given instant in time.

Conceptually, each node maintains a set of pending events. At each step in the execution, the model checker selects one of the nodes and an event pending at that node. The model checker then runs the appropriate event handler to transition the system to a new state. The handler may send messages that get added to event queues of destination nodes or schedule timers to add more events to its pending set. Upon completing an event handler, control returns to the model checker and we repeat the process. Each program execution corresponds to a scheduling of interleaved events and a sequence of transitions.

Properties A *state predicate* is a logical predicate over the program variables. Each state predicate evaluates to or in any given state. We say that a state *satisfies* (resp., *violates*) a state predicate if the predicate evaluates to (resp.,) in the state.

Safety Property: a statement of the form *always* p where p is a *safety (state) predicate*. An execution *satisfies* a safety property if *every* state in the execution satisfies p. Conversely, an execution *violates* a safety property if *some* state in the execution violates p.

Liveness Property: a statement of the form *always eventually* p where p is a *liveness (state) predicate*. We define program states to be in exactly one of three categories with respect to a liveness property: *live*, *dead*, or *transient*. A live state satisfies p. A transient state does not satisfy p, but some execution through the state leads to a live state. A dead state does not satisfy p, and no execution through the state leads to a live state. An execution *satisfies* a liveness property if every suffix of the execution contains a live state. In other words, an execution satisfies the liveness property if the system enters a live state infinitely often during the execution. Conversely, an execution *violates* a liveness property if the execution has a suffix without any live states.

System	Name		Property
Pastry	AllNodes	*Eventually*	$\forall n \in \textbf{nodes} : n.(successor)^* \equiv \textbf{nodes}$ Test that all nodes are reached by following successor pointers from each node.
	SizeMatch	*Always*	$\forall n \in \textbf{nodes} : n.myright.size() + n.myleft.size() = n.myleafset.size()$ Test the sanity of the leafset size compared to left and right set sizes.
Chord	AllNodes	*Eventually*	$\forall n \in \textbf{nodes} : n.(successor)^* \equiv \textbf{nodes}$ Test that all nodes are reached by following successor pointers from each node.
	SuccPred	*Always*	$\forall n \in \textbf{nodes} : \{n.predecessor = n.me \iff n.successor = n.me\}$ Test that a node's predecessor is itself if and only if its successor is itself.
RandTree	OneRoot	*Eventually*	**for** *exactly* 1 $n \in \textbf{nodes} : n.isRoot$ Test that exactly one node believes itself to be the root node.
	Timers	*Always*	$\forall n \in \textbf{nodes} : \{(n.state = init) \| (n.recovery.nextScheduled() \neq 0)\}$ Test that either the node state is $init$, or the recovery timer is scheduled.
MaceTransport	AllAcked	*Eventually*	$\forall n \in \textbf{nodes} : n.inflightSize() = 0$ Test that no messages are in-flight (i.e., not acknowledged).
			No corresponding safety property identified.

Table 1: Example predicates from systems tested using MACEMC. *Eventually* refers here to *Always Eventually* corresponding to Liveness properties, and *Always* corresponds to Safety properties. The syntax allows a regular expression expansion '*', used in the AllNodes property.

It is important to stress that liveness properties, unlike safety properties, apply over entire program executions rather than individual states. Classically, states cannot be called live (only executions)—we use the term live state for clarity. The intuition behind the definition of liveness properties is that any violation of a liveness state predicate should only be temporary: in any live execution, regardless of some violating states, there must be a future state in the execution satisfying the liveness predicate.

Table 1 shows example predicates from systems we have tested in MACEMC. We use the same liveness predicate for PASTRY and CHORD, as both form rings with successor pointers.

Liveness/Safety Duality We divide executions violating liveness into two categories: Transient-state and Dead-state. *Transient-state (TS) liveness violations* correspond to executions with a suffix containing only transient states. For example, consider a system comprising two servers and a randomized job scheduling process. The liveness property is that eventually, the cumulative load should be balanced between the servers. In one TS liveness violation, the job scheduling process repeatedly prefers one server over the other. Along a resulting infinite execution, the cumulative load is never balanced. However, at every point along this execution, it is possible for the system to recover, e.g., the scheduler could have balanced the load by giving enough jobs to the underutilized server. Thus, all violating states are transient and the system never enters a dead state.

Dead-state (DS) liveness violations correspond to an execution with any dead state (by definition all states following a dead state must also be dead because recovery is impossible). Here, the violating execution takes a *critical transition* from the last transient (or live) state to the first dead state. For example, when checking an overlay tree (cf. §6), we found a violating execution of the "OneRoot" liveness state predicate in Table 1, in which two trees formed independently and never merged. The critical transition incorrectly left the $recovery$ timer of a node A unscheduled in the presence of disjoint trees. Because only A had knowledge of members in the other tree, the protocol had no means to recover.

Our work focuses on finding DS liveness violations. We could have found these violations by using safety properties specifying that the system never enters the corresponding dead states. Unfortunately, these safety properties are often impossible to identify *a priori*. For instance, consider the liveness property "AllNodes" for CHORD shown in Table 1: eventually, all nodes should be reachable by following successor pointers. We found a violation of this property caused by our failure to maintain the invariant that in a one-node ring, a node's predecessor and successor should be itself. Upon finding this error, we added the corresponding safety property for CHORD. While we now see this as an "obvious" safety property, we argue that exhaustively listing all such safety properties *a priori* is much more difficult than specifying desirable liveness properties.

Moreover, liveness properties can identify errors that in practice are infeasible to find using safety properties. Consider the "AllAcked" property for our implementation of a transport protocol, shown in Table 1. The property is for the test application, which sends a configurable total number of messages to a destination. It states that all sent messages should eventually be acknowledged by the destination (assuming no permanent failures): the transport adds a message to the $inflight$ queue upon sending and removes it when it is acknowledged. The corresponding safety property would have to capture the following: "Always, for each message in the *inflight* queue or retransmission timer queue, either the message is in flight (in the network), or in the destination's receive socket buffer, or the receiver's corresponding $IncomingConnection.next$ is less than the message sequence number, or an acknowledgment is in flight from the destination to the sender with a sequence number greater than or equal to the message sequence number, or the same acknowledgment is in the sender's receive socket buffer, or a reset message is in flight between the sender and receiver (in either direction), or ..." Thus, attempting to specify certain conditions with safety properties quickly becomes overwhelming and hopelessly complicated, especially when contrasted with the simplicity and succinctness of the liveness property: "Eventually, for all n in nodes, $n.inflightSize() = 0$," i.e., that eventually there should be no packets in flight.

Thus, we recommend the following iterative process for finding subtle protocol errors in complex concurrent environments. A developer begins by writing desirable high-level liveness properties. As these liveness properties typically define the correct system behavior in steady-state operation, they are relatively easy to specify. Developers can then leverage insight from DS liveness violations to add new safety properties. In Table 1, we show safety properties that became apparent while analyzing the corresponding DS liveness violations. While safety properties are often less intuitive, the errors they catch are typically easier to understand—the bugs usually do not involve complex global state and lie close to the operations that trigger the violations.

3 Model Checking with MACEMC

This section presents our algorithms for finding liveness and safety violations in systems implementations. We find potential liveness violations via a three-step state exploration process. While our techniques do not present proofs for the existence of a liveness violation, we have thus far observed no false positives. In practice, all flagged violations must be human-verified, which is reasonable since they point to bugs which must be fixed. As shown in Figure 1, our process isolates executions leading the system to dead states where recovery to a configuration satisfying the liveness state predicate becomes impossible.

Step 1: Bounded depth-first search (BDFS) We begin by searching from an initial state with a bounded depth-first search. We exhaustively explore all executions up to some fixed depth in a depth-first manner and then repeat with an increased depth bound. Due to state explosion, we can only exhaustively explore up to a relatively shallow depth of transitions (on the order of 25-30); as system initialization typically takes many more transitions (cf. Figure 2), the vast majority of states reached at the periphery of the exhaustive search are not live. We call these states *indeterminate* because at this point we do not yet know whether they are dead or transient.

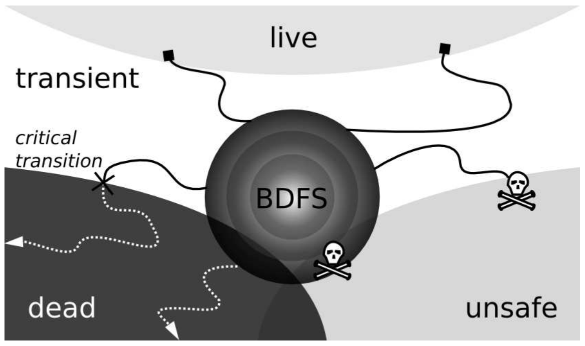


Figure 1: **State Exploration** We perform bounded depth-first search (BDFS) from the initial state (or search prefix): most periphery states are indeterminate, i.e., not live, and thus are either dead or transient. We execute random walks from the periphery states and flag walks not reaching live states as suspected violating executions.

Step 2: Random Walks While the exhaustive search is essential to finding a candidate set of liveness violations, to prune the false positives, we must distinguish the dead from the transient states. To do so, we perform long random walks to give the system sufficient time to enter a live state. If the system still fails to reach a live state by the end of the walk, we flag the execution as a suspected liveness violation. Our random walks typically span tens or hundreds of thousands of transitions to minimize the likelihood of false positives.

Step 3: Isolating the Critical Transition The model checker presents the execution exhibiting a suspected liveness violation to the developer to assist in locating the actual error. The programmer cannot understand the bug simply by examining the first states that are not live, as these are almost always transient states, i.e., there exist executions that would transition these initial indeterminate states to live states. Thus, we developed an algorithm to automatically isolate the *critical transition* that irreversibly moves the system from a transient state to a dead state.

3.1 Finding Violating Executions

We now describe the details of our algorithms. Suppose that MACEMC is given a system, a safety property *always* p_s, and a liveness property *eventually* p_l.

Our algorithm MaceMC_Search (Algorithm 1) systematically explores the space of possible executions. Each execution is characterized by the sequence of choices

Algorithm 1 MaceMC_Search

```
Input: Depth $increment$
  $depth = 0$
  repeat
    if $Sequences(depth)$ is empty then
      $depth = depth + increment$
    Reset system
    $seq$ = next sequence in $Sequences(depth)$
    MaceMC_Simulator($seq$)
  until STOPPING CONDITION
```

made to determine the node-event pair to be executed at each step. We iterate over all the sequences of choices of some fixed length and explore the states visited in the execution resulting from the sequence of choices. Consider the set of all executions bounded to a given depth $depth$. These executions form a tree by branching whenever one execution makes a different choice from another. To determine the order of executions, we simply perform a depth-first traversal over the tree formed by this depth bound. $Sequences(depth)$ returns a sequence of integers indicating which child to follow in the tree during the execution. It starts by returning a sequence of 0's, and each time it is called it increases the sequence, searching all possible sequences. For each sequence, MaceMC_Search initializes the system by resetting the values of all nodes' variables to their initial values and then calls the procedure MaceMC_Simulator to explore the states visited along the execution corresponding to the sequence. After searching all sequences of length $depth$, we repeat with sequences of increasing depth. We cannot search extreme system depths due to the exponential growth in state space. While they have not been necessary to date, optimizations such as multiple random walks or best-first search may enhance coverage over initial system states.

Algorithm 2 MaceMC_Simulator

```
Input: Sequence $seq$ of integers
  for $i = 0$ to $d_{max}$ do
    $readyEvents$ = set of pending $\langle node, event \rangle$ pairs
    $eventnum$ = Toss($i, seq, |readyEvents|$)
    $\langle node, event \rangle = readyEvents[eventnum]$
    Simulate $event$ on $node$
    if $p_s$ is violated then
      signal SAFETY VIOLATION
    if $i > depth$ and $p_l$ is satisfied then
      return
  signal SUSPECTED LIVENESS VIOLATION
```

Algorithm 2, MaceMC_Simulator, takes a sequence of integers as input and simulates the resulting execution using the sequence of choices corresponding to the integers. MaceMC_Simulator simulates an execution of up to d_{max} transitions (cf. §4.4 for setting d_{max}). At the i^{th} step, MaceMC_Simulator calls the procedure Toss with i, the sequence, and the number of ready events to determine pending node event pairs to execute, and then executes the handler for the chosen event on the chosen node to obtain the state reached after i transitions. If this state violates the given safety predicate, then MaceMC_Simulator reports the safety violation. If this state is beyond the search depth and satisfies the given liveness predicate, then the execution has not violated the liveness property and the algorithm returns. Only considering liveness for states beyond the search depth is important because otherwise a live state within the periphery would prevent us from finding liveness bugs that enter the dead state beyond the periphery. If the loop terminates after d_{max} steps, then we return the execution as a suspected liveness violation.

Combining Exhaustive Search and Random Walks The procedure Toss ensures that MaceMC_Search and MaceMC_Simulator together have the effect of exhaustively searching all executions of bounded depths and then performing random walks from the periphery of the states reached in the exhaustive search. Toss(i, seq, k) returns the i^{th} element of the sequence seq if i is less than $|seq|$ (the length of the sequence) or some random number between 0 and k otherwise. Thus, for the first $|seq|$ iterations, MaceMC_Simulator selects the $seq[i]^{th}$ element of the set of pending node event pairs, thereby ensuring that we exhaustively search the space of all executions of depth $|seq|$. Upon reaching the end of the supplied sequence, the execution corresponds to a random walk of length $d_{max} - |seq|$ performed from the periphery of the exhaustive search. By ensuring d_{max} is large enough (hundreds of thousands of transitions), we can give the system enough opportunity to reach a live state. If the execution never enters a live state despite this opportunity, we flag the execution as a suspected liveness violation.

3.2 Finding the Critical Transition

If MACEMC reaches the maximum random walk depth d_{max} without entering a live state, we have a suspected liveness violation. The execution meets one of two conditions:

*Condition 1 (*C1*):* The execution is a DS liveness violation, meaning the system will never recover. The execution should be brought to the attention of the programmer to locate and fix the error.

*Condition 2 (*C2*):* The execution does not reach any live states, but might still in the future. The execution should be brought to the attention of the programmer to determine whether to proceed by increasing d_{max} or by inspecting the execution for a bug.

Before discussing how we distinguish between the two cases, consider an execution that does enter a dead state (meets condition C1). The programmer now faces the daunting and time consuming task of wading through tens of thousands of events to isolate the protocol or implementation error that transitioned the system to a dead state. Recall that while the system may enter a transient state early, typically a much later critical transition finally pushes the system into a dead state. After attempting to find liveness errors manually when only the violating execution was available, we set out to develop an algorithm to automatically locate the critical transition. Importantly, this same procedure also heuristically identifies whether an execution meets C1 or C2.

Algorithm 3 FindCriticalTransition

```
Input: Execution E non-live from step $d_{init}$ to $d_{max}$
Input: Number of Random Walks k
Output: (Critical Transition $d_{crit}$, Condition C1 or C2)
1:  {Phase 1: Exponential Search}
2:  if not Recovers($E$, $d_{init}, k$) then return ($d_{init}$,C2)
3:  $d_{curr} = d_{init}$
4:  repeat
5:      $d_{prev} = d_{curr}$
6:      $d_{curr} = 2 \times d_{curr}$
7:      if $d_{curr} > d_{max}/2$ then return ($d_{curr}$,C2)
8:  until not Recovers($E$, $d_{curr}, k$)
9:  {Phase 2: Binary Search}
10: {$d_{prev}$ is highest known recoverable}
11: {$d_{curr}$ is lowest believed irrecoverable}
12: loop
13:     if ($d_{prev} = d_{curr} - 1$) then return ($d_{curr}$,C1)
14:     $d_{mid} = (d_{prev} + d_{curr})/2$
15:     if Recovers($E$, $d_{mid}, k$) then $d_{prev} = d_{mid}$
16:     else $d_{curr} = d_{mid}$
```

Algorithm 3 shows our two-phase method for locating the critical transition. It takes as input the execution *E* from the initial random walk, which from step d_{init} onwards never reached a live state even after executing to the maximum depth d_{max}. The function Recovers(E, i, k) performs up to *k* random walks starting from the *i*th state on the execution *E* to the depth d_{max} and returns if *any* of these walks hit a live state, indicating that the *i*th state should be marked transient; and otherwise, indicating that the *i*th state is dead. In the first phase, MACEMC doubles d_{curr} until Recovers indicates that d_{curr} is dead. d_{max} and the resulting d_{curr} place an upper bound on the critical transition, and the known live state d_{prev} serves as a lower bound. In the second phase, MACEMC performs a binary search using Recovers to find the critical transition as the *first* dead state d_{crit} between d_{prev} and d_{curr}. If we perform *k* random walks from each state along the execution, then the above procedure takes $O(k \cdot d_{max} \cdot \log d_{crit})$ time (Note that $d_{crit} \leq d_{max}$).

In addition to the full execution that left the system in a dead state and the critical transition d_{crit}, we also present to the programmer the event sequence that shares the longest common prefix with the DS liveness violation that ended in a live state. In our experience, the combination of knowing the critical transition and comparing it to a similar execution that achieves liveness is invaluable in finding the actual error.

Two interesting corner cases arise in the FindCriticalTransition algorithm. The first case occurs when Phase 1 cannot locate a dead state (indicated by $d_{curr} > d_{max}/2$ in line 7). In this case, we conclude that as the critical transition does not appear early enough, the system was not given enough opportunity to recover during the random walk. Thus, case C2 holds. The developer should raise d_{max} and repeat. If raising d_{max} does not resolve the problem, the developer should consider the possibility that this execution is a TS liveness violation. To help this analysis, MACEMC provides the set of live executions similar to the violating execution, but the developer must isolate the problem. In the second case, we find no live executions even when in the initial state (line 2); either the critical transition is at d_{init} (the initial state), or, more likely, we did not set d_{max} high enough. The programmer can typically determine with ease whether the system condition at d_{init} contains a bug. If not, once again we conclude that case C2 holds and raise d_{max} and repeat Algorithm 1.

4 Implementation Details

This section describes several subtle details in our MACEMC implementation. While we believe the techniques described in Section 3 could be applied to any state-exploration model checker capable of replaying executions, MACEMC operates on systems implemented using the MACE compiler and C++ language extensions [18]. MACE introduces syntax to structure each node as a state machine with atomic handlers corresponding to events such as message reception, timers firing, etc. MACE implementations consist of C++ code in appropriately identified code blocks describing system state variables and event handler methods; and the MACE compiler outputs C++ code ready to run across the Internet by generating classes and methods to handle event dispatch, serialization, timers, callbacks, etc. MACE implementations perform comparably or better than hand-tuned implementations. Leveraging MACE code frees us from the laborious task of modifying source code to isolate the execution of the system, e.g., to control network communication events, time, and other sources of potential input. Thus, using MACE-implemented systems dramatically improves the accessibility of model checking to the typical programmer.

4.1 Preparing the System

To model check a system, the user writes a driver application suitable for model checking that should initialize the system, perform desired system input events, and check high-level system progress with liveness properties. For example, to look for bugs in a file distribution protocol, the test driver could have one node supply the file, and the remaining nodes request the file. The liveness property would then require that all nodes have received the file and the file contents match. Or for a consensus protocol, a simulated driver could propose a different value from each node, and the liveness property would be that each node eventually chooses a value and that all chosen values match. The MACEMC application links with the simulated driver, the user's compiled MACE object files, and MACE libraries. MACEMC simulates a distributed environment to execute the system—loading different simulator-specific libraries for random number generation, timer scheduling, and message transport—to explore a variety of event orderings for a particular system state and input condition.

Non-determinism To exhaustively and correctly explore different event orderings of the system, we must ensure that the model checker controls all sources of non-determinism. So far, we have assumed that the scheduling of pending ⟨node, event⟩ pairs accounts for all non-determinism, but real systems often exhibit non-determinism *within* the event handlers themselves, due to, e.g., randomized algorithms and comparing timestamps. When being model checked, MACE systems automatically use the deterministic simulated random number generator provided by MACEMC and the support for simulated time, which we discuss below. Furthermore, we use special implementations of the MACE libraries that internally call Toss at every non-deterministic choice point. For example, the TCP transport service uses Toss to decide whether to break a socket connection, the UDP transport service uses Toss to determine which message to deliver (allowing out-of-order messages) and when to drop messages, and the application simulator uses Toss to determine whether to reset a node. Thus, by systematically exploring the *sequences* of return values of Toss (as described in MaceMC_Search in the previous section), MACEMC analyzes all different sequences of internal non-deterministic choices. Additionally, this allows MACEMC to deterministically replay executions for a given sequence of choices.

Time Time introduces non-determinism, resulting in executions that may not be replayable or, worse, impossible in practice. For example, a system may branch based on the relative value of timestamps (e.g., for message timeout). But if the model checker were to use actual values of time returned by im a , this comparison might always be forced along one branch as the simulator fires events faster than a live execution. Thus, MACEMC must represent time abstractly enough to permit exhaustive exploration, yet concretely enough to only explore feasible executions. In addition, MACEMC requires that executions be deterministically replayable by supplying an identical sequence of chosen numbers for all non-deterministic operations, including calls to im a .

We observed that systems tend to use time to: (i) manage the passage of real time, e.g., to compare two timestamps when deciding whether a timeout should occur, or, (ii) export the equivalent of monotonically increasing sequence numbers, e.g., to uniquely order a single node's messages. Therefore, we address the problem of managing time by introducing two new MACE object primitives— a im and im — to obtain and compare time values. When running across a real network, both objects are wrappers around im a . However, MACEMC treats every comparison between a im objects as a call to Toss and implements im objects with counters. Developers concerned with negative clock adjustments (and more generally non-monotone im implementations) can strictly use a im to avoid missing bugs, at the cost of extra states to explore. Compared to state of the art model checkers, this approach frees developers from manually replacing time-based non-determinism with calls to Toss, while limiting the amount of needless non-determinism.

4.2 Mitigating State Explosion

One stumbling block for model-checking systems is the exponential explosion of the state space as the search depth increases. MACEMC mitigates this problem using four techniques to find bugs deep in the search space.

1. Structured Transitions The event-driven, non-blocking nature of MACE code significantly simplifies the task of model-checking MACE implementations and improves its effectiveness. In the worst case, a model checker would have to check all possible orderings of the assembler instructions across nodes with pending events, which would make it impractical to explore more than a few hundred lines of code across a small number of nodes. Model checkers must develop techniques for identifying larger atomic steps. Some use manual marking, while others interpose communication primitives. Non-blocking, atomic event handlers in MACE allow us to use event-handler code blocks as the fundamental unit of execution. Once a given code block runs to completion, we return control to MACEMC. At this point, MACEMC checks for violations of any safety or liveness conditions based on global system state.

2. State Hashing When the code associated with a particular event handler completes without a violation, MACEMC calculates a hash of the resulting system state. This state consists of the concatenation of the values of all per-node state variables and the contents of all pending, system-wide events. The programmer may optionally annotate MACE code to ignore the value of state variables believed to not contribute meaningfully to the uniqueness of global system state, or to format the string representation into a canonical form to avoid unneeded state explosion (such as the order of elements in a set). MaceMC_Simulator checks the hash of a newly-entered state against all previous state hashes. When it finds a duplicate hash, MACEMC breaks out of the current execution and begins the next sequence. In our experience, this allows MACEMC to avoid long random walks for 50-90 percent of all executions, yielding speedups of 2-10.

3. Stateless Search MACEMC performs backtracking by re-executing the system with the sequence of choices used to reach an earlier state, similar to the approach taken by Verisoft [11]. For example, to backtrack from the system state characterized by the sequence $\langle 0, 4, 0 \rangle$ to a subsequent system state characterized by choosing the sequence $\langle 0, 4, 1 \rangle$, MACEMC reruns the system from its initial state, re-executing the event handlers that correspond to choosing events 0 and 4 before moving to a different portion of the state space by choosing the event associated with value 1. This approach is simple to implement and does not require storing all of the necessary state (stack, heap, registers) to restore the program to an intermediate state. However, it incurs additional CPU overhead to re-execute system states previously explored. We have found trading additional CPU for memory in this manner to be reasonable because CPU time has not proven to be a limitation in isolating bugs for MACEMC. However, the stateless approach is not fundamental to MACEMC—we are presently exploring hybrid approaches that involve storing some state such as sequences for best-first searching or state for checkpointing and restoring system states to save CPU time.

4. Prefix-based Search Searching from an initial global state suffers the drawback of not reaching significantly past initialization for the distributed systems we consider. Further, failures during the initial join phase do not have the opportunity to exercise code paths dealing with failures in normal operation because they simply look like an aborted join attempt (e.g., resulting from dropped messages) followed by a retry. To find violations in steady-state system operation, we run MACEMC to output a number of live executions of sufficient length, i.e., executions where all liveness conditions have been satisfied, all nodes have joined, and the system has entered steady-state operation. We then proceed as normal from one of these live prefixes with exhaustive searches for safety violations followed by random walks from the perimeter to isolate and verify liveness violations. We found the PASTRY bug described in the introduction using a prefix-based search.

4.3 Biasing Random Walks

We found that choosing among the set of all possible actions with equal probability had two undesirable consequences. First, the returned error paths had unlikely event sequences that obfuscated the real cause of the violation. For example, the system generated a sequence where the same timer fired seven times in a row with no intervening events, which would be unlikely in reality. Second, these unlikely sequences slowed system progress, requiring longer random walks to reach a live state. Setting d_{max} large enough to ensure that we had allowed enough time to reach live states slowed FindCriticalTransition by at least a factor of ten.

We therefore modified Toss to take a set of weights corresponding to the rough likelihood of each event occurring in practice. Toss returns an event chosen randomly with the corresponding probabilities. For example, we may prioritize application events higher than message arrivals, and message arrivals higher than timers firing. In this way, we *bias* the system to search event sequences in the random walk with the hope of reaching a live state sooner, if possible, and making the error paths easier to understand.

Biasing the random walks to common sequences may run counter to the intuition that model checkers should push the system into corner conditions difficult to predict or reason about. However, recall that we run random walks only after performing exhaustive searches to a certain depth. Thus, the states reached by the periphery of the exhaustive search encompass many of these tricky corner cases, and the system has already started on a path leading to—or has even entered—a dead state.

One downside to this approach is that the programmer must set the relative weights for different types of events. In our experience, however, every event has had a straightforward rough relative probability weighting. Further, the reductions in average depth before transitioning to a live state and the ease of understanding the violating executions returned by MACEMC have been worthwhile. If setting the weights proves challenging for a particular system, MACEMC can be run with unbiased random walks.

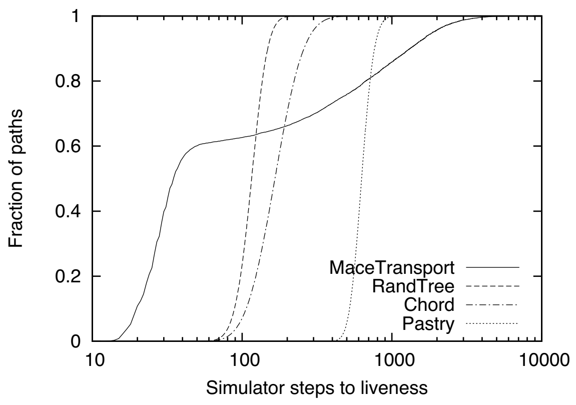


Figure 2: CDF of simulator steps to a live state at a search depth of 15.

4.4 Tuning MACEMC

In addition to event weights discussed above, MACEMC may be tuned by setting d_{max} (random walk depth), k (number of random walks), and a wide variety of knobs turning features on and off. Feature knobs include whether to test node failures, socket failures, UDP drops, UDP reordering, and the number of simulated nodes, and are generally easy to set based on the target test environment.

Setting k is a bit more complex. k represents the tradeoff between the time to complete the critical transition algorithm and the possibility that the reported critical transition is before the actual critical transition. This occurs when k random executions of d_{max} steps did not satisfy liveness, but some other path could have. We informally refer to this occurrence as "near dead". In our tests, we general use k between 20 and 60. At 60, we have not observed any prematurely reported critical transitions, while at 20 we occasionally observe the reported critical transition off by up to 2 steps. To tune k, the programmer considers the output critical transition. If it is not obvious why it is the critical transition, the programmer can increase k and re-run to refine the results.

Finally, we discuss how to set d_{max}. We ran MACEMC over four systems using random walks to sample the state space beyond an exhaustive search to 15 steps. Figure 2 plots the fraction of executions that reached the first live state at a given depth. What we observe is that in these four systems, since all sample executions reached a live state by 10,000 steps, a random execution that takes 80,000 steps to reach a live state would be a significant outlier, and likely somewhere along the execution it became trapped in a region of dead states. Setting d_{max} too low generally leads to the critical transition algorithm reporting condition C2, which is what we treat as the signal to increase d_{max}.

Figure 2 also illustrates that the depths required to initially reach a live state are much greater than what can be found with exhaustive search. MACEMC found only 60% of executions reached a live state for MACE-TRANSPORT after considering 50 steps (the edge of what can be exhaustively searched using state-of-the-art model checkers), less than 1% of executions for RANDTREE and CHORD, and none of the executions for PASTRY.

5 MACEMC Debugger

Although MACEMC flags violating executions and identifies the critical transition that likely led the system to a dead state, the developer must still understand the sequence of events to determine the root cause of the error. This process typically involves manually inspecting the log files and hand-drawing sketches of evolving system state. To simplify this process, we built MDB, our debugging tool with support for interactive execution, replay, log analysis, and visualization of system state across individual nodes and transitions. MDB is similar in function to other work in distributed debuggers such as the WiDS Checker [22] and Friday [10]. MDB allows the programmer to: (i) perform single step system execution both forward and backward, (ii) jump to a particular step, (iii) branch execution from a step to explore a different path, (iv) run to liveness, (v) select a specific node and step through events only for that node, (vi) list all the steps where a particular event occurred, (vii) filter the log using regular expressions, and (viii) *diff* the states between two steps or the same step across different executions by comparing against a second, similar log file.

MDB also generates event graphs that depict inter-node communication. It orders the graph by nodes on the x-axis and simulator steps on the y-axis. Each entry in the graph describes a simulated event, including the transition call stack and all message fields. Directional arrows represent message transmissions, and other visual cues highlight dropped messages, node failures, etc.

MDB recreates the system state by analyzing detailed log files produced by MACEMC. While searching for violations, MACEMC runs with all system logging disabled for maximum efficiency. Upon discovering a violation, MACEMC automatically replays the path with full logging. The resulting log consists of annotations: (i) written by the programmer, (ii) generated automatically by the MACE compiler marking the beginning and end of each transition, (iii) produced by the simulator runtime libraries, such as timer scheduling and message queuing and delivery, and (iv) generated by the simulator to track the progress of the run, including random number requests and results, the node simulated at each step, and the state of the entire system after each step. For our runs, logs can span millions of entries (hundreds to thousands of

```
   ./mdb error.log
 m       j 5
 m       filediff live.log
...
localaddress=2.0.0.1:10201
out=[
− OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=2, packetsSent=2, acksReceived=1, packetsRetransmitted=0),
−        inflight=[ 6002 → MessageInfo(seq=6002, syn=0, retries=0, timeout=true) ],
−        rtbuf=[ ], sendbuf=[ ], curseq=6002, dupacks=0, last=6001)
+ OutgoingConnection(1.0.0.1:10201, connection=ConnectionInfo(cwnd=1, packetsSent=1, acksReceived=0, packetsRetransmitted=0),
+        inflight=[ 6001 → MessageInfo(seq=6001, syn=1, retries=0, timeout=true) ],
+        rtbuf=[ ], sendbuf=[ MessageInfo(seq=6002, syn=0, timer=0, retries=0, timeout=true) ], curseq=6002, dupacks=0, last=0)
]
in=[ ]
−timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=6002, syn=0, retries=0, timeout=true)])
+timer<retransmissionTimer>([dest=1.0.0.1:10201, msg=MessageInfo(seq=6001, syn=1, retries=0, timeout=true)])
...
```

Figure 3: MDB session. Lines with differences are shown in italics (− indicates the error log, + the live log), with differing text shown in bold. The receiver is IP address 1.0.0.1 and the sender is 2.0.0.1.

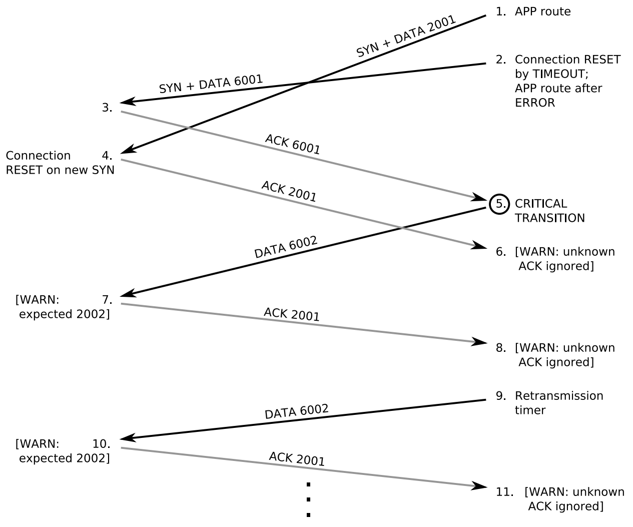

1. APP route
SYN + DATA 2001
2. Connection RESET by TIMEOUT; APP route after ERROR
SYN + DATA 6001
3.
ACK 6001
Connection RESET on new SYN 4.
ACK 2001
5. CRITICAL TRANSITION
DATA 6002
6. [WARN: unknown ACK ignored]
[WARN: expected 2002] 7.
ACK 2001
8. [WARN: unknown ACK ignored]
9. Retransmission timer
DATA 6002
[WARN: expected 2002] 10.
ACK 2001
11. [WARN: unknown ACK ignored]

Figure 4: Automatically generated event graph for MACETRANSPORT liveness bug.

megabytes).

To demonstrate the utility of our debugging tools for diagnosing and fixing errors, we consider a case study with a bug in MACETRANSPORT: a reliable, in-order, message delivery transport with duplicate-suppression and TCP-friendly congestion-control built over UDP. Unlike TCP, MACETRANSPORT is fundamentally message- rather than stream-oriented, making it a better match for certain higher-level application semantics. As such, rather than using sequence numbers to denote byte offsets as with TCP, MACETRANSPORT assigns an incrementing sequence number to each packet. To obtain lower-latency communication, MACETRANSPORT avoids a three-way handshake to establish initial sequence numbers. A key high-level liveness property for MACETRANSPORT is that eventually every message should be acknowledged (unless the connection closes).

MACEMC found a violating execution of the "AllAcked" property in Table 1, where a sender attempts to send two messages to a receiver. Figure 4 shows a pictorial version of the event graphs automatically generated by MDB; the actual event graph is text-based for convenience and contains more detail. In Step 1, the sender sends a data packet with the SYN flag set and sequence number 2001. In Step 2, the retransmission timer causes the connection to close and MACETRANSPORT signals an error to the application. The application responds by attempting to resend the packet, causing MACETRANSPORT to open a new connection with sequence number 6001. At this point, both the old "SYN 2001" and the new "SYN 6001" packets are in flight. In Step 3, the network delivers the packet for the new 6001 connection, and the receiver replies by sending an "ACK 6001" message. In Step 4, the network delivers the out-of-order "SYN 2001" message, and the receiver responds by closing the connection on 6001, thinking it is stale, and opening a new incoming connection for 2001.

Unfortunately, in Step 5 (the critical transition) the sender receives the "ACK 6001." Believing the 6000-sequence connection to be established, the sender transmits "DATA 6002," at odds with the receiver's view. From here on, the execution states are dead as the receiver keeps ignoring the "DATA 6002" packet, sending ACKs for the 2001 connection instead, while the sender continues to retransmit the "DATA 6002" packet, believing it to be the sequence number for the established connection.

We illustrate a portion of an MDB session analyzing this bug in Figure 3. We load the error log in MDB, jump to the critical transition step (5), and *diff* the state with the live path with the longest shared prefix (output by MACEMC while searching for the critical transition (see §3.2)). The excerpt shows the state for the sender node. The key insight from this output is that in the live execution (lines indicated with $+$), the retransmission timer is scheduled with "SYN 6001," meaning that the packet could be retransmitted and the receiver could become resynchronized with the sender. Comparing the differences with the violating execution (lines indicated with $-$), where 6001 has been removed from the *inflight* map and timer because of the ACK, allows us to identify and fix the bug by attaching a monotonically increasing identifier in the SYN packets, implemented using a im object. Now, when the receiver gets the "SYN 2001" message out of order, it correctly concludes from the identifier that the message is stale and should be ignored, allowing acknowledgment of the "DATA 6002" message.

6 Experiences

We have used MaceMC to find safety and liveness bugs in a variety of systems implemented in Mace, including a reliable transport protocol, an overlay tree, Pastry, and Chord. With the exception of Chord, we ran MaceMC over mature implementations manually debugged both in local- and wide-area settings. MaceMC found several subtle bugs in each system that caused violations of high-level liveness properties. All violations (save some found in Chord, see below) were beyond the scope of existing software model checkers because the errors manifested themselves at depths far beyond what can be exhaustively searched. We used the debugging process with Chord as control—we first performed manual debugging of a new implementation of Chord and then employed MaceMC to compare the set of bugs found through manual and automated debugging.

Table 2 summarizes the bugs found with MaceMC to date. This includes 52 bugs found in four systems. Spanning the three mature systems, the 33 bugs across 1500 lines of Mace code correspond to one bug for every 50 lines of code. MaceMC actually checks the generated C++ code, corresponding to one bug for every 250 lines of code. In the only comparable check of a complex distributed system, CMC found approximately one bug for every 300 lines of code in three versions of the AODV routing protocol [25]. Interestingly, more than 50% of the bugs found by CMC were memory handling errors (22/40 according to Table 4 [25]) and all were safety violations. The fact that MaceMC finds nearly the same rate of errors while focusing on an entirely different class of liveness errors demonstrates the complementary nature of the bugs found by checking for liveness rather than safety violations. To demonstrate the nature and complexity of liveness violations we detail two representative violations below; we leave a detailed discussion of each bug we found to a technical report [17].

System	Bugs	Liveness	Safety	LOC
MaceTransport	11	5	6	585/3200
RandTree	17	12	5	309/2000
Pastry	5	5	0	621/3300
Chord	19	9	10	254/2200
Totals	52	31	21	

Table 2: Summary of bugs found for each system. LOC=Lines of code and reflects both the Mace code size and the generated C++ code size.

Typical MaceMC run times in our tests have been from less than a second to a few days. The median time for the search algorithm has been about 5 minutes. Typical critical-transition algorithm runtimes are from 1 minute to 3 hours, with the median time being about 9 minutes.

RandTree implements a random overlay tree with a maximum degree designed to be resilient to node failures and network partitions. This tree forms the backbone for a number of higher-level aggregation and gossip services including our implementations of Bullet [21] and RanSub [20]. We have run RandTree across emulated and real wide-area networks for three years, working out most of the initial protocol errors.

RandTree nodes send a "Join" message to a bootstrap node, who in turn forwards the request up the tree to the root. Each node then forwards the request randomly down the tree to find a node with available capacity to take on a new child. The new parent adds the requesting node to its child set and opens a TCP connection to the child. A "JoinReply" message from parent to child confirms the new relationship.

Property. A critical high-level liveness property for RandTree (and other overlay tree implementations) is that all nodes should eventually become part of a single spanning tree.

We use four separate Mace liveness properties to capture this intuition: (i) there are no loops when following $parent$ pointers, (ii) a node is either the root or has a parent, (iii) there is only one root (shown in Table 1), and (iv) each node N's parent maintains it as a child, and N's children believe N to be their parent.

Violation. MaceMC found a liveness violation where two nodes A, D have a node C in their child set, even though C's parent pointer refers to D. Along the violating execution, C initially tries to join the tree under B, which forwards the request to A. A accepts C as a child and sends it a "JoinReply" message. Before establishing the connection, C experiences a node reset, losing all state. A, however, now establishes the prior connection with the new C, which receives the "JoinReply' and ignores it (having been reinitialized). Node C then attempts to join

the tree but this time is routed to D, who accepts C as a child. Node A assumes that if the TCP socket to C does not break, the child has received the "JoinReply" message and therefore does not perform any recovery. Thus, C forever remains in the child sets of A and D.
Bug. The critical transition for this execution is the step where C receives the "JoinReply" from A. MDB reveals that upon receiving the message, C ignores the message completely, without sending a "Remove" message to A. Along the longest live *alternate* path found from the state prior to the critical transition, we find that instead of receiving A's join reply message, C gets a request from the higher-level application asking it to join the overlay network, which causes C to transition into a "joining" mode from its previous "init" mode. In this alternate path, C subsequently receives A's "JoinReply" message, and correctly handles it by sending A a "Remove" message. Thus, we deduced that the bug was in C's ignoring of "JoinReply" messages when in the "init" mode. We fix the problem by ensuring that a "Remove" reply is sent in this mode as well.

CHORD specifies a key-based routing protocol [30]. CHORD structures an overlay in a ring such that nodes have pointers to their successor and predecessor in the key-space. To join the overlay a new node gets its predecessor and successor from another node. A node inserts itself in the ring by telling its successor to update its predecessor pointer, and a stabilize procedure ensures global successor and predecessor pointers are correct through each node probing its successor.
Property. We use a liveness property to specify that all nodes should eventually become part of a single ring (see Table 1). This minimal correctness condition guarantees that routing reach the correct node.
Violation. MACEMC found a liveness violation in the very first path it considered. This was not unexpected, given that CHORD had not been tested yet. However, the critical transition algorithm returned transition 0 and condition C2, implying that the algorithm could not determine if the path had run long enough to reach liveness.

Looking at the event graph, we saw the nodes finished their initial join quickly (step 11), and spent the remaining steps performing periodic recovery. This process suggested that the system as a whole was dead, since reaching a live state would probably not require tens of thousands of transitions when the initial join took only 11.

MDB showed us that mid-way through the execution, client0's successor pointer was client0 (implying that it believed it was in a ring of size 1), which caused the liveness predicate to fail. The other nodes' successor pointers correctly followed from client1 to client2 to client0. We believed the $stabilize$ procedure should correct this situation, expecting client2 to discover that client0 (its successor) was in a self-loop and correct the situation. Looking at this procedure in the event graph, we saw that there was indeed a probe from client2 to client0. However, client2 ignored the response to this probe. We next jumped to the transition in MDB corresponding to the probe response from the event graph. In fact, client0 reported that client2 was its predecessor, so client2 did not correct the error.

Starting at the initial state in MDB we stepped through client0's transitions, checking its state after each step to see when the error symptom occurs. After 5 steps, client0 receives a message that causes it to update its predecessor but *not* its successor, thus causing the bug.

Bug. This problem arose because we based our original implementation of CHORD on the original protocol [30], where a joining node explicitly notified its predecessor that it had joined. We then updated our implementation to the revised protocol [31], which eliminated this notification and specified that all routing state should be updated upon learning of a new node. However, while we removed the join notification in our revisions, we failed to implement the new requirements for updating routing state, which we overlooked because it concerned a seemingly unrelated piece of code. We fixed the bug by correctly implementing the new protocol description.

Overall, both our manual testing and model checking approaches found slightly different sets of bugs. On the one hand, manual testing found many of the correctness bugs and also fixed several performance issues (which cannot be found using MACEMC). Manual testing required that we spend at least half of our time trying to determine whether or not an error even occurred. A single application failure may have been caused by an artifact of the experiment, or simply the fact that the liveness properties had not yet been satisfied. Because of these complexities, identifying errors by hand took anywhere from 30 minutes to several hours per bug.

On the other hand, MACEMC did find some additional correctness bugs and moreover required less human time to locate the errors. MACEMC examines the state-snapshot across all nodes after each atomic event and reports only known bugs, thereby eliminating the guesswork of determining whether an error actually occurred. Furthermore, the model checker outputs which property failed and exactly how to reproduce the circumstances of the failure. MACEMC also produces a verbose log and event graph, and in the case of liveness violations, an alternate path which would have been successful. These features make it much easier to verify and identify bugs using MACEMC, without the hassle of conducting experiments that require running many hosts on a network. We spent only 10 minutes to an hour using MACEMC to find the same bugs that we painstakingly identified earlier with manual testing; and we found the new bugs (those not caught with manual testing) in only tens of minutes.

7 Related Work

Our work is related to several techniques for finding errors in software systems that fall under the broad umbrella of Model Checking.

Classical Model Checking. "Model Checking," i.e., checking a system described as a graph (or a Kripke structure) was a model of a temporal logic formula independently invented in [6, 27]. Advances like *Symmetry Reduction, Partial-Order Reduction*, and *Symbolic Model Checking* have enabled the practical analysis of hardware circuits [2, 23], cache-coherence and cryptographic protocols [9], and distributed systems and communications protocols [15], which introduced the idea of state-hashing used by MACEMC. However, the tools described above require the analyzed software to be specified in a tool-specific language, using the state graph of the system constructed either before or during the analysis. Thus, while they are excellent for quickly finding *specification* errors early in the design cycle, it is difficult to use them to verify the systems *implementations*. MACEMC by contrast tests the C++ implementation directly, finding bugs both in the design and the implementation.

Model Checking by Random Walks. West [32] proposed the idea of using random walks to analyze networking protocols whose state spaces were too large for exhaustive search. Sivaraj and Gopalakrishnan [29] propose a method for iterating exhaustive search and random walks to find bugs in cache-coherence protocols. Both of the above were applied to check safety properties in systems described using specialized languages yielding finite state systems. In contrast, MACEMC uses random walks to find liveness bugs by classifying states as dead or transient, and further, to pinpoint the critical transition.

Model Checking by Systematic Execution. Two model checkers that directly analyze implementations written in C and C++ are VERISOFT [11] and CMC [25]. VERISOFT views the entire system as several *processes* communicating through message queues, semaphores and shared variables *visible* to VERISOFT. It schedules these processes and traps calls that access shared resources. By choosing the process to execute at each such trap point, the scheduler can exhaustively explore all possible interleavings of the processes' executions. In addition, it performs stateless search and partial order reduction allowing it to find critical errors in a variety of complex programs. Unfortunately, when we used VERISOFT to model-check MACE services, it was unable to exploit the atomicity of MACE's transitions, and this combined with the stateless search meant that it was unable to exhaustively search to the depths required to find the bugs MACEMC found. A more recent approach, CMC [25], also directly executes the code and explores different executions by interposing at the scheduler level. CMC has found errors in implementations of network protocols [24] and file systems [34]. JAVAPATHFINDER [14] takes an approach similar to CMC for JAVA programs. Unlike VERISOFT, CMC, and JAVAPATHFINDER, MACEMC addresses the challenges of finding liveness violations in systems code and simplifying the task of isolating the cause of a violation.

Model Checking by Abstraction. A different approach to model checking software implementations is to first *abstract* them to obtain a finite-state model of the program, which is then explored exhaustively [3, 4, 7, 8, 12, 16] or up to a bounded depth using a SAT-solver [5, 33]. Of the above, only FEAVER and BANDERA can be used for liveness-checking of concurrent programs, and they require a user to manually specify how to abstract the program into a finite-state model.

Isolating Causes from Violations. Naik et al. [26] and Groce [13] propose ways to isolate the cause of a safety violation by computing the difference between a violating run and the closest non-violating one. MACEMC instead uses a combination of random walks and binary search to isolate the critical transition causing a liveness violation, and then uses a live path with a common prefix to help the programmer understand the root cause of the bug.

8 Conclusions

The most insidious bugs in complex distributed systems are those that occur after some unpredictable sequence of asynchronous interactions and failures. Such bugs are difficult to reproduce—let alone fix—and typically manifest themselves as executions where the system is unable to *ever* enter some desired state after an error occurs. In other words, these bugs correspond to violations of *liveness* properties that capture the designer's intention of how the system should behave in steady-state operation. Though prior software model checkers have dramatically improved our ability to find and eliminate errors, elusive bugs like the subtle error we found in PASTRY have been beyond their reach, as they only find violations of *safety* properties.

We have described techniques that enable software model checkers to heuristically isolate the complex bugs that cause liveness violations in systems implementations. A key insight behind our work is that many interesting liveness violations correspond to the system entering a *dead* state, from which recovery to the desired state is impossible. Though a safety property describing dead states exists mathematically, it is often too complex and implementation-specific for the programmer to specify without knowing the exact bug in the first place. Thus, we have found that the process of finding the errors that cause liveness violations often reveals previously unknown safety properties, which can be used to find and fix more errors. We have used MACEMC to find 31 liveness (and 21 safety) errors in MACE implementations of four complex distributed systems. We believe that our techniques—a combination of state-exploration, random walks, critical transition identification, and MDB—radically expand the scope of implementation model checkers to include liveness violations, thereby enabling programmers to isolate subtle errors in systems implementations.

Acknowledgements

We would like to thank our shepherd, Petros Maniatis, for his many insights and contributions to this paper, and our anonymous reviewers, for their valuable comments.

References

[1] Freepastry: an open-source implementation of pastry intended for deployment in the internet. h r a r ri , 2006.

[2] ALUR, R., HENZINGER, T., MANG, F., QADEER, S., RAJAMANI, S., AND TASIRAN, S. MOCHA: modularity in model checking. In *Computer-aided Verification (CAV)*, A. Hu and M. Vardi, Eds., Lecture Notes in Computer Science 1427. Springer-Verlag, 1998, pp. 521–525.

[3] BALL, T., AND RAJAMANI, S. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)* (2002).

[4] CHEN, H., AND WAGNER, D. MOPS: an infrastructure for examining security properties of software. In *Computer and Communications Security (CCS)* (2002).

[5] CLARKE, E., KROENING, D., AND LERDA, F. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)* (2004), K. Jensen and A. Podelski, Eds., vol. 2988 of *Lecture Notes in Computer Science*, pp. 168–176.

[6] CLARKE, E. M., AND EMERSON, E. A. Synthesis of synchronization skeletons for branching time temporal logic. In *Logic of Programs* (1981), Lecture Notes in Computer Science 131.

[7] COOK, B., PODELSKI, A., AND RYBALCHENKO, A. Termination proofs for systems code. In *Programming Language Design and Implementation (PLDI)* (2006).

[8] CORBETT, J., DWYER, M., HATCLIFF, J., PASAREANU, C., ROBBY, LAUBACH, S., AND ZHENG, H. Bandera : Extracting finite-state models from Java source code. In *International Conference on Software Engineering (ICSE)* (2000).

[9] DILL, D., DREXLER, A., HU, A., AND YANG, C. H. Protocol verification as a hardware design aid. In *International Conference on Computer Design (ICCD)* (1992).

[10] GEELS, D., ALTEKAR, G., MANIATIS, P., ROSCOE, T., AND STOICA, I. Friday: Global comprehension for distributed replay. In *Networked Systems Design and Implementation (NSDI)* (2007).

[11] GODEFROID, P. Model checking for programming languages using Verisoft. In *Principles of Programming Languages (POPL)* (1997).

[12] GRAF, S., AND SAÏDI, H. Construction of abstract state graphs with PVS. In *Computer-aided Verification (CAV)*, Lecture Notes in Computer Science 1254. 1997, pp. 72–83.

[13] GROCE, A., AND VISSER, W. What went wrong: Explaining counterexamples. In *Spin Model Checking and Software Verification (SPIN)* (2003).

[14] HAVELUND, K., AND PRESSBURGER, T. Model checking Java programs using Java Pathfinder. *Software Tools for Technology Transfer (STTT) 2(4)* (2000), 72–84.

[15] HOLZMANN, G. The Spin model checker. *Transactions on Software Engineering 23*, 5 (1997), 279–295.

[16] HOLZMANN, G. Logic verification of ANSI-C code with SPIN. In *Spin Model Checking and Software Verification (SPIN)* (2000), Lecture Notes in Computer Science 1885.

[17] KILLIAN, C., ANDERSON, J., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. Tech. rep., University of California, San Diego. h ma a r a _ .

[18] KILLIAN, C., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. Mace: Language support for building distributed systems. In *Programming Languages Design and Implementation (PLDI)* (2007).

[19] KINDLER, E. Safety and liveness properties: A survey. *EATCS-Bulletin*, 53 (1994).

[20] KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., BHIRUD, A., AND VAHDAT, A. Using Random Subsets to Build Scalable Network Services. In *USENIX Symposium on Internet Technologies and Systems (USITS)* (2003).

[21] KOSTIĆ, D., RODRIGUEZ, A., ALBRECHT, J., AND VAHDAT, A. Bullet: High bandwidth data dissemination using an overlay mesh. In *Symposium on Operating Systems Principles (SOSP)* (2003).

[22] LUI, X., LIN, W., PAN, A., AND ZHANG, Z. Wids checker: Combating bugs in distributed systems. In *Networked Systems Design and Implementation (NSDI)* (2007).

[23] MCMILLAN, K. L. A methodology for hardware verification using compositional model checking. *Science of Computer Programming 37*, (1–3) (2000), 279–309.

[24] MUSUVATHI, M., AND ENGLER, D. R. Model checking large network protocol implementations. In *Networked Systems Design and Implementation (NSDI)* (2004).

[25] MUSUVATHI, M., PARK, D., CHOU, A., ENGLER, D., AND DILL, D. CMC: A pragmatic approach to model checking real code. In *Operating Systems Design and Implementation (OSDI)* (2002).

[26] NAIK, M., BALL, T., AND RAJAMANI, S. From symptom to cause: Localizing errors in counterexample traces. In *Principles of Programming Languages (POPL)* (2003).

[27] QUEILLE, J., AND SIFAKIS, J. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, M. Dezani-Ciancaglini and U. Montanari, Eds., Lecture Notes in Computer Science 137. Springer-Verlag, 1981.

[28] ROWSTRON, A., AND DRUSCHEL, P. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Middleware* (2001).

[29] SIVARAJ, H., AND GOPALAKRISHNAN, G. Random walk based heuristic algorithms for distributed memory model checking. *Electr. Notes Theor. Comput. Sci. 89*, 1 (2003).

[30] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, F., AND BALAKRISHNAN, H. Chord: A scalable peer to peer lookup service for internet applications. In *ACM Special Interest Group on Data Communication (SIGCOMM)* (2001).

[31] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Transactions on Networking 11*, 1 (2003), 17–32.

[32] WEST, C. H. Protocol validation by random state exploration. In *6th IFIP WG 6.1 International Workshop on Protocol Specification, Testing, and Verification* (1986).

[33] XIE, Y., AND AIKEN, A. Scalable error detection using boolean satisfiability. In *Principles of Programming Languages (POPL)* (2005).

[34] YANG, J., TWOHEY, P., ENGLER, D. R., AND MUSUVATHI, M. Using model checking to find serious file system errors. In *Operating Systems Design and Implementation (OSDI)* (2004).

WiDS Checker: Combating Bugs in Distributed Systems

Xuezheng Liu
Microsoft Research Asia

Wei Lin
Microsoft Research Asia

Aimin Pan
Microsoft Research Asia

Zheng Zhang
Microsoft Research Asia

Abstract

Despite many efforts, the predominant practice of debugging a distributed system is still printf-based log mining, which is both tedious and error-prone. In this paper, we present WiDS Checker, a unified framework that can check distributed systems through both simulation and reproduced runs from real deployment. All instances of a distributed system can be executed within one simulation process, multiplexed properly to observe the "happens-before" relationship, thus accurately reveal full system state. A versatile script language allows a developer to refine system properties into straightforward assertions, which the checker inspects for violations. Combining these two components, we are able to check distributed properties that are otherwise impossible to check. We applied WiDS Checker over a suite of complex and real systems and found non-trivial bugs, including one in a previously proven Paxos specification. Our experience demonstrates the usefulness of the checker and allows us to gain insights beneficial to future research in this area.

1 Introduction

From large clusters in machine rooms to large-scale P2P networks, distributed systems are at the heart of today's Internet services. At the same time, it is well recognized that these systems are difficult to design, implement, and test. Their protocols involve complex interactions among a collection of networked machines, and must handle failures ranging from network problems to crashing nodes. Intricate sequences of events can trigger complex errors as a result of mishandled corner cases. The most challenging bugs are not the ones that will crash the system immediately, but the ones that corrupt certain design properties and drive the system to unexpected behaviors after long runs.

Yet the predominant practice in debugging distributed systems has remained unchanged over the years: manually inspecting logs dumped at different machines. Typically, developers embed printf statements at various implementation points, perform tests, somehow stitch the logs together, and then look for inconsistencies. However, log mining is both labor-intensive and fragile. Log events are enormous in number, making the inspection tedious and error-prone. Latent bugs often affect application properties that are themselves distributed across multiple nodes, and verifying them from local events can be very difficult. More important, logs reflect only incomplete information of an execution, sometimes insufficient to reveal bugs. Pip [23], for instance, logs the application behavior in terms of communication structures, timing and resource usage, and compares them against developer expectations. However, our experience shows that applications with correct message sequences can perform wrong things and mutate inner states because of buggy logic. Therefore, it is impossible to catch the existence of these subtle bugs only from communication logs, unless much more state is also logged.

It is a common experience that omitting a key logging point can miss a bug thus defeating the entire debugging exercise, yet adding it could substantially change subsequent runs. The non-determinism of distributed applications plus the limitations of log-based debugging makes such "Heisenbugs" a nightmare for developers. Building a time machine so that bugs can be deterministically replayed gets rid of the artifacts of using logs [8]. However, one still lacks a comprehensive framework to express correctness properties, catch violation points, and identify root causes.

We believe that a desired debugging tool for distributed applications needs to: 1) efficiently verify application properties, especially distributed ones; 2) provide fairly complete information about an execution, so that developers can observe arbitrary application states, rather than pre-defined logs; 3) reproduce the buggy runs deterministically and faithfully, and hence enable the cyclic debugging process.

In this paper, we address the above debugging requirements with a unified framework called WiDS Checker. This platform logs the actual execution of a distributed system implemented using the WiDS toolkit [15]. We can then apply predicate checking in a centralized simulator over a run that is either driven by testing scripts or is deterministically replayed by the logs. The checker outputs violation reports along with message traces, allowing us to perform "time-travel" inside the Visual Studio IDE to identify the root causes.

1.1 Our Results

Evaluating the effectiveness of our tool is a challenge. The research community, though acutely aware of the difficulties of debugging distributed systems, has not succeeded in producing a comprehensive set of benchmarks so that different debugging approaches can be quantitatively compared. While we believe the set of applications we have experimented with are representative, there is also no clear methodology of how to quantify the usefulness of the tool. To alleviate these problems, we resort to detailed discussions of case studies, hoping that other researchers working on similar systems can compare their experiences. Where possible, we also isolate the benefits coming from predicate checking from using log and replay only. Our tool is targeted at the scenario in which the system is debugged by those who developed it, and thus assumes that the bugs are hunted by those who are intimately familiar with the system. How to propagate the benefits to others who are not as versed in the system itself is an interesting research question.

We applied WiDS Checker to a suite of distributed systems, including both individual protocols and a complete, deployed system. Within a few weeks, we have found non-trivial bugs in all of them. We discovered both deadlock and livelock in the distributed lock service of Boxwood [19]. We checked an implementation of the Chord protocol [1] on top of Macedon [24] and found five bugs, three of them quite subtle. We checked our BitVault storage system [32], a deployed research prototype being incrementally improved for more than two years. We identified mishandling of race conditions that can cause loss of replicas, and incorrect assumptions of transient failures. The most interesting experience was checking our Paxos [13] implementation, revealing a bug in a well-studied specification itself [21]. Most of these bugs were not discovered before; All our findings are confirmed by the authors or developers of the corresponding systems.

We also learned some lessons. Some bugs have deep paths and appear only at fairly large scale. They cannot be identified when the system is downscaled, thus calling for more efficient handling of the state explosion problem when a model checker is applied to check actual implementation. Most of the bug cases we found have correct communication structure and messages. Therefore, previous work that relies on verifying event ordering is unable to detect these bugs, and is arguably more effective for performance bugs.

1.2 Paper Roadmap

The rest of the paper is organized as follows. Section 2 gives an overview of the checker architecture. Section 3 provides implementation details. Section 4 presents our results, including our debugging experience and lessons learned. Section 5 contains related work and we conclude with future work in Section 6.

2 Methodology and Architecture Overview

2.1 Replay-Based Predicate Checking

The WiDS[1] checker is built on top of the WiDS toolkit, which defines a set of APIs that developers use to write generic distributed applications (details see Section 3.1). Without modification, a WiDS-based implementation can be simulated in a single simulation process, simulated on a cluster-based parallel simulation engine, or deployed and run in real environment. This is made possible by linking the application binary to three different runtime libraries (simulation, parallel simulation and deployment) that implement the same API interface. WiDS was originally developed for large-scale P2P applications; its parallel simulation engine [14] has simulated up to 2 million instances of a product-strength P2P protocol, and revealed bugs that only occur at scale [29]. With a set of basic fault injection utilities, WiDS allows a system to be well tested inside its simulation-based testing framework before its release to deployment. WiDS is available with full source code in [3].

However, it is impossible to root out all bugs inside the simulator. The deployed environment can embody different system assumptions, and the full state is unfolded unpredictably. Tracking bugs becomes extremely challenging, especially for the ones causing violation of system properties that are themselves distributed. When debugging non-distributed software and stand-alone components, developers generally check memory states against design-specified correctness properties at runtime using invariant predicates (e.g., *assert()* in C++). This dynamic predicate checking technique is proven of great help to debugging; many advanced program-checking tools are effective for finding domain-specific bugs (e.g., race conditions [25, 31] and memory leaks [22, 10]) based on the same principle. However, this benefit does not extend to distributed systems for two reasons. First, distributed properties reside on multiple machines and cannot be directly evaluated at one place without significant runtime perturbations. Second, even if we can catch a violation, the cyclic debugging process is broken because non-determinism across runs makes it next to impossible to repeat the same code path that leads to the bug.

To address this problem and to provide similar checking capabilities to distributed systems, we propose a **replay-based predicate checking** approach that allows the execution of the entire system to be replayed afterwards within a single machine, and at the same time checks node states during the replayed execution against user-defined predicates. Under modest scale, this solves both problems outlined above.

2.2 Checking Methodology

User-defined predicates are checked at *event* granularity. An event can be an expiration of a timer, receiving a message from another node, or scheduling and synchronization events (e.g., resuming/yielding a thread and acquiring/releasing a lock) specific for thread programming. WiDS interprets an execution of a single node or the entire distributed system as a sequence of events, which are dispatched to corresponding handling routines. During the replay, previous executed events from all nodes are re-dispatched, ordered according to the "happens-before" relationship [12]. This way the entire system is replayed in the simulator while preserving causality.

Each time an event is dispatched, the checker evaluates predicates and reports violations for the current step in replay. The choice of using event boundaries for predicate checking is due to a number of factors. First, the event model is the basis of many protocol specifications, especially the ones based on I/O-automata [18, 17]. A system built with the event model can be regarded as a set of state machines in which each event causes a state transition executed as an atomic step. Distributed properties thus change at event boundaries. Second, many widely adopted implementation models can be distilled into such a model. We have used the WiDS toolkit [15] to build large and deployed systems as well as many protocols; network middle-layers such as Macedon [24] and new overlay models such as P2 [16] can be regarded as event-based platforms as well. We believe event granularity is not only efficient, but also sufficient.

2.3 Architecture

Figure 1 shows the architecture of WiDS Checker.

Reproducing real runs. When the system is running across multiple machines, the runtime logs all non-deterministic events, including messages received from

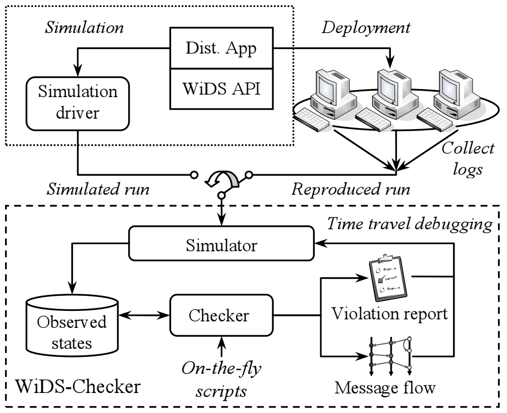


Figure 1: Components of WiDS Checker. The upper-left box was employed as debugging support in WiDS before the Checker was developed.

network, data read from files, thread scheduling decisions and many environmental system calls. The executable binary is then rerun inside the simulator, and all non-deterministic events are fed from the logs. We use Lamport's logical clock [12] to decide the replay order of events from different nodes, so that the "happens-before" relationship is preserved. Therefore, inside the simulator, we reconstruct the exact state of all instances as they are run in the real environment.

Checking user-defined predicates. We designed a versatile scripting language to specify system states being observed and the predicates for invariants and correctness. After each step of event-handling, the observed states are retrieved from the replayed instance in the simulator, and refreshed in a database. Then, the checker evaluates predicates based on the current states from all replayed instances and reports violations. Because predicates generally reflect design properties, they are easy to reason and write.

Screening out false alarms with auxiliary information. Unlike safety properties, liveness properties are only guaranteed to be true eventually. This poses a serious problem when checking liveness properties, since many violations can be false alarms. To screen out false alarms, we enable user-defined auxiliary information to be calculated and output along with each violation point. When a violation occurs, the auxiliary information is typically used to produce stability measures based on user-provided heuristics.

Tracing root causes using a visualization tool. In addition to the violation report, we generate a message flow graph based on event traces. All these facilities are integrated into the Visual Studio Integrated Development Environment (IDE). Thus, a developer can "time-travel" to violation points, and then trace backwards while inspecting the full state to identify root causes.

The first two components make it possible to check distributed properties, whereas the last two are critical features for a productive debugging experience. Note that developers can incrementally refine predicates and re-evaluate them on the same reproduced execution. In other words, by means of replay, cyclic debugging is re-enabled.

3 Design and Implementation

WiDS Checker depends critically on the replay functionality, which is done at the API level. In this section, we will first briefly describe these APIs. We then explain the replay facility and the checker implementation. This section concludes with a discussion of the known limitations. WiDS has close to 20K lines of code. The replay and checker components add 4K and 7K lines, respectively.

3.1 Programming with WiDS

Table 1 lists the class of WiDS APIs with some examples. The WiDS APIs are mostly member functions of the *WiDSObject* class, which typically implements one node instance of a distributed system. The WiDS runtime maintains an event queue to buffer pending events and dispatches them to corresponding handling routines (e.g., *OnMsgHandler()*). Beside this event-driven model, WiDS also supports multi-threaded programming with its thread and synchronization APIs. The context switching of WiDS threads is encapsulated as events in the event queue. We use non-preemptive scheduling in which the scheduling points are WiDS API and blocking system calls, similar to many user-level thread implementations. The fault-injection utilities include dropping or changing the latency of messages, and killing and restarting WiDS objects.

Macedon over WiDS. It's not easy to write a distributed application with message-passing and event-driven model. Defining better language support is an active research area [16, 24]. WiDS can be used as the low-level mechanism to implement such languages, and thus make its full functionality available to their applications. As an experiment, we ported Macedon [24], an infrastructure to specify and generate implementation of overlay protocols. Macedon has been used to reimplement many complex overlay protocols with a simple domain language and conduct performance comparisons. Porting to the WiDS API is simplified because both Macedon and WiDS provide a set of APIs working in an event-driven manner, and programming entities such as messages and timers exist in both platforms. An overlay protocol expressed with Macedon is composed of a .mac file, which our parser takes as input to generate a set of WiDS implementation files. Many overlay specific functionalities provided by the Macedon library are replicated in a WiDS-based library. Finally, all these files are linked to generate an executable, which, with a proper driver, can be both simulated and deployed. Supporting Macedon is accomplished with about 4K lines of code.

3.2 Enabling replay

The primary goal of the deterministic replay is to reproduce the exact application memory states inside the simulator. To achieve this, we need to log all non-deterministic inputs to the application and feed them to the replay simulator.

Logging. The WiDS runtime logs the following two classes of non-determinism: The first is internal to WiDS. We record all the WiDS events, the WiDS thread schedule decisions and the incoming message content. The second class are OS system calls, including reading from files, returned memory addresses for allocation and deallocation in heap, and miscellaneous others such as system time and random number generation. In Windows NT, each API call is redirected by the linker to the import address table (IAT), from which another jump is taken to reach the real API function. We changed the address in the IAT, so the second jump will lead to the appropriate logging wrapper, which will log the return results after the real API is executed. Furthermore, to enable consistent group replay [8], we embed a Lamport Clock [12] in each out-going message's header to perserve the "happens-before" relation during the replay. Table 1 describes logging and replay mechanisms for API calls.

In addition, we use a lightweight compressor to effectively reduce the log size. As we report in Section 4.5, the computation overhead for logging is small in the tests we performed.

Checkpoint. We use checkpoints to avoid over-committing storage overhead from logging and to support partial replay during replay. A checkpoint includes the snapshot of memory of the WiDS process and the running context for user-level threads and sockets as well as buffered events in the event queue.

Replay. Replaying can start from either the beginning or a checkpoint. Note that checking predicates requires all instances to be replayed with causality among them preserved. Therefore, during the replay, events from different instances are collected from logs, sequentialized into a total execution order based on the Lamport Clock, and re-executed one-by-one in the simulator.

To improve replay performance and scalability, we use only *one* simulation process to replay all instances, and use file-mapping to deal with the memory switch between different instances. The state of an instance is

WiDS API set		
Category	API example	Logging and replay mechanism
Event-driven program	SetTimer, KillTimer, OnTimerExpire	Log the event type and the sequence; redo the same events in replay
Message communication	PostMsg, PostReliableMsg, OnMsgHandler	Embed Lamport Clock to maintain causal order, log incoming message contents. Replay with correct partial order, feed message content.
Multi-threaded program	CreateThread, JoinThread, KillThread, YieldThread, Lock, Unlock	Log the schedule decision and the thread context. Ensure the same schedule decision and the same context during replay
Socket APIs for network virtualization	WiDSSocket, WiDSListen, WiDSAccept, WiDSConnect, WiDSSend, WiDSRecv	Log the operation along with all received data. Feed the received data from log during replay. Sending operations become no-ops in replay
Fault injection and message delay	ActivateNode, DeActivateNode, SetNetworkModel, OnCalculateDelay	Log the operation of activation/deactivation, and redo the operation in replay
Operation system APIs (for Windows)		
File system	CreateFile, OpenFile, ReadFile, WriteFile, CloseHandle, SetFilePointer	Log the operation along with all input data. Feed the input data from log during replay. Write operations become no-ops in replay
Memory management	VirtualAlloc/Free, HeapAlloc/Free	Ensure identical memory layout in replay.
Miscellaneous	GetSystemTimeAsFileTime, GetLastError	Log the return value, and feed the same value in replay

Table 1: WiDS API set and OS APIs with logging and replay mechanisms

stored in a memory mapped file, and is mapped into the process memory space on-demand. For example, to switch the replayed instance from A to B, we only update the entries in the page table of the simulation process to the base address of the mapped memory of B. Starting a process for each replayed instance and switching the processes would require local process communication (LPC) that is typically tens of times slower than function calls. Therefore, our approach has significant advantages. When the aggregated working set of all replayed instances fit into physical memory, the only overhead in switching instance is the update to the page table. It also avoids redundant memory usage caused by process context and executable binary for each instance.

The largest per-instance working set in our experiments is about 20MB, meaning that more than 40 instances can be replayed in 1GB physical memory without going to the disk. When the aggregated working set exceeds the physical memory, depending on the computation density of the replayed instance, we have observed 10 to 100 times slowdown due to disk swapping. The checker itself maintains a copy of the states being checked, and that varies across applications. Ultimately, the scalability is bound by the disk size and acceptable replay speed.

3.3 Checker

Deterministic replay that properly preserves causality has enabled the reconstruction of memory states of a distributed system. The next step is to write predicate statements to catch the violation points of correctness properties. We define a simple scripting language for specifying predicates, so that developers can easily specify the structure of the investigated states, retrieve them from the memory of the instances, and evaluate properties from these states.

As mentioned before, checking predicates is invoked at event boundaries. Each time an event is re-executed in a replayed instance, the checker examines the state changes in the instances, and re-evaluates the affected predicates. The states being checked are copies kept in a separate database. The checker refreshes these states in the database from the replayed instance and evaluates predicates based on the state copies of all instances. Therefore, checking predicates is decoupled from memory layout of the instances, and we do not require all instances to reside in memory simultaneously for evaluating global properties. This makes replay and the checker more scalable. Furthermore, maintaining state copies separately allows us to keep past versions of states if needed, which is useful for evaluating certain properties (see Section 4.1).

In this section, we explain the checker implementation, including the necessary reflection facilities that make memory states in C++ objects observable by the checker, state maintenance and predicate evaluation, and the auxiliary information associated with violations that deal with false alarms.

3.3.1 Observing memory states

For programming languages such as Java and C# that support runtime reflection, the type system and user-defined data structures are observable during the runtime. Unfortunately, this is not the case for C++, which is what WiDS uses. To check application states, we need to record the memory address of each allocated C++ object with type information during its lifetime. We use the Phoenix compiler infrastructure [2] to analyze the executable and inject our code to track class types and object addresses. Phoenix provides compiler-independent intermediate representation of binary code, from which we are able to list basic blocks, function calls, and the symbol table that contains all type definitions. We then inject our logging function to function calls of constructors and deconstructors of the classes. The logging function will dump the timestamp, type of operation (i.e., construction or deconstruction) along with the address of object and type information. This information is used by the checker to inspect memory states. The following assembly codes show an example of a constructor after code injection. The lines beginning with "*" are injected code. They call our injected logging function *onConstruct* with the index number of this class found in symbol table. We perform similar code injection for object deconstruction. As a result, at each step of the replay, the checker is capable of enumerating pointers of all objects of a certain class, and further reading their memory fields according to the symbol table. The runtime overhead is negligible since the actions are only triggered at object allocation and deallocation time.

```
$L1: (refs=0)  START MyClass::MyClass
MyClass::MyClass: (refs=1)
this = ENTERFUNC
 * [ESP], {ESP} = push 0x17 //index number for MyClass
 * call _imp__onConstruct@4, $out[ESP] //call log func
[ESP], {ESP} = push EBP
EBP = mov ESP [ESP],
{ESP} = push ECX
... // other code in original constructor
ESP = mov EBP EBP,
{ESP} = pop [ESP]
{ESP} = ret {ESP}, MyClass::MyClass
MyClass::MyClass: (refs=1) Offset: 32(0x0020)
EXITFUNC
$L2: (refs=0) END
```

This code injection is carried out in a fully automated way. In addition, we provide some APIs that allow developers to explicitly calculate and expose states of an instance in the source code.

3.3.2 Defining states and evaluating predicates

A script for predicate evaluation is composed of three parts: declaration of tables, declaration of internal variables, and the predicates. Figure 2 shows the script we used for checking the Chord protocol [28] implemented on Macedon (see Section 4.4 for details).

The first section (starting with "**declare_table**") instructs the checker to observe objects of some classes and refresh the states of certain member fields into the database. Each row of the table corresponds to one object in the system, and table columns correspond to member fields of the object. Each table has two built-in columns *instance_id* and *memory_addr*, corresponding to the replayed instance and the object's memory address, respectively. The declaration gives the user shorthand notations to name the table and the states. A table stores global states from all instances, e.g., the table *Node* here maintains the investigated states of all the Chord nodes in the system. Sometimes it is useful to keep a short history of a state for checking. We provide an optional **keep_version**(N) after a column declaration to declare that the recent N versions of the state should be kept in the table.

```
                        # define data table
declare_table Node from CChord
   column id as m_nodeid
   column pred as m_predecessor
   column succ as m_successor
   column status as m_status
end_declare
                        # define checker variables
declare_derived last_churn_time
begin_python
   for x in Node :
         if  (x.status == 0            # status "0" means joining
            or Runtime.msd_id== 108);         # MSG_FAIL_NOTIFY
            return Runtime.current_time;
   return last_churn_time;
end_python

declare_derived stabilized
begin_python
   retval = (Runtime.current_time - last_churn_time) / 10.0;
          if (retval < 1) : return retval;
   return 1;
end_python
                        # define predicates
predicate RingConsistency auxiliary stabilized{
   forall x in Node, exist y in Node,
      x.pred==y.id and y.succ == x.id
}
```

Figure 2: An example of check scripts for chord. The auxiliary information *Stabilized* is reset to 0 when joins or failures occur; otherwise it gradually grows to 1.

The second section allows users to define variables internal to the checker with the keyword **declare_derived**. These variables can also have histories, again using **keep_version**(N). Between **begin_python** and **end_python** are python snippets to calculate the value of a named variable. The python snippet has read access to values of all prior declarations (i.e., data tables and internal variables), using the declared names. Data tables are regarded as enumerable python containers, indexed by a (*instance_id*, *memory_addr*) pair.

The last section uses the keyword "**predicate**" to specify correctness properties based on all declared states and varables, which are evaluated after refreshing tables and after the evaluation of internal variables. Each predicate is a Boolean expression. We support the set of common logical operators, e.g., **and**, **or**, **imply**. We also support two quantifiers, **forall** and **exist**, which specify the extent of validity of a predicate when dealing with tables. These built-in operators make it easy to specify many useful invariants. In Figure 2, the predicate states that the ring should be well formed: if node x believes node y to be its predecessor, then y must regard x as its successor. It is a critical property for the stabilization of Chord topology.

After each step of the replay, the checker does the following. First, it enumerates all objects of classes defined in data tables in the memory of a replayed instance. It uses the type information and memory address provided by the log to refresh the table, inserting or deleting rows, and updating the columns accordingly. After updating tables, the checker also knows which declared states have changed, and it only re-evaluats all the affected derived values and predicates. When some predicates are evaluated as "false," the checker outputs the violation into a *violation report*, which contains the violated predicates, Lamport Clock value for each violation, and the auxiliary information defined in the script (discussed shortly).

Sometimes it is useful to replay and check a segment of execution, rather than to do it from the beginning. The problem here is how to reconstruct the states maintained by checker scripts when the checkpoint is loaded. To solve this problem, we support checkpoints in replay runs, which store both replay context and the tables and variables used by predicate scripts. These replay checkpoints can be used seamlessly for later checking. To start checking with an intermediate checkpoint from testing runs, the developers have to provide additional scripts to setup the states required by the script from the memory of instances in the checkpoint. Otherwise, there might be incorrect predicate evaluations caused by checkpoints.

3.3.3 Auxiliary information for violations

For safety properties that must hold all the time, every violation reveals a bug case. In contrast, liveness properties only guarantee to be true eventually, so a violation of liveness properties is not necessarily a bug case. For example, many overlay network systems employ self-stabilizing protocols to deal with churns, therefore most of their topology-related properties are liveness ones. As a result, checking liveness properties could generate a large number of false alarms that overwhelm the real violations. Adding a time bound to liveness properties is not always a desirable solution, because usually it's hard to derive an appropriate time bound.

To solve the problem, we enable users to attach auxiliary information to the predicates. The auxiliary information is a user-defined varable calculated along with the predicate, and it is only output when the predicate is violated. Developers could used the information to help screen out false alarms or prioritize violations. For liveness properties, an appropriate usage for auxiliary information is to output a measurement of a stabilization condition. For example, in Figure 2 we associate the eventual ring consistency property with an auxiliray variable *Stabilized* ranging from 0 to 1, as a measure of stabilization that shows the "confidence" of the violation.

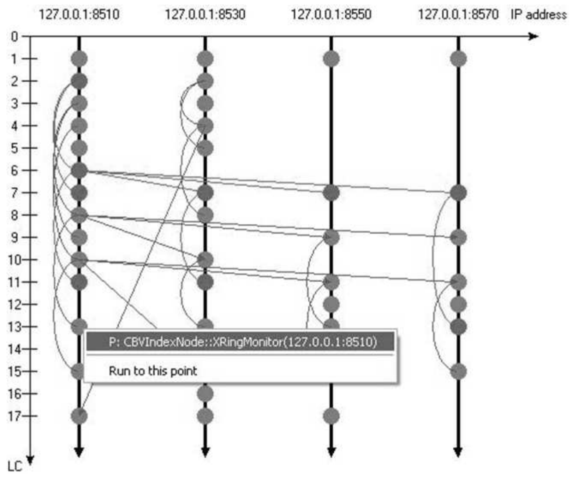


Figure 3: A screenshot of message flow graph. The vertical lines represent the histories of different instance, arcs denote messages across instances, and the nodes correspond to event handlings. Arcs with two ends on the same vertical lines are either timer events or messages sending to the instance itself.

We also maintain some built-in system parameters in the checker, e.g., the current time in the node, the current message type, and statistics of recent messages of each type. These parameters can be directly accessed in the scripts, and are useful in stabilization measurement. Our evaluation section contains more examples of using the auxiliary information.

3.4 Visualization tools

To pinpoint the root cause of a bug, a user often needs to trace back in time from a violation point. In addition to our replay facility, we provide the message flow graph generated based on message trace (Figure 3) to make this process easier. It is a common practice in our experience to perform time travel following the message flow, replay to a selected event point and inspect memory state of the replayed instance. We find that the visualization helps us understand system behaviors as well as the root cause after catching violations.

4 Evaluation

In this section, we report our experience of finding bugs using WiDS Checker over a comprehensive set of real systems. Table 2 gives a summary of the results. The checking scripts to reveal the bugs are short and easy to write from system properties. For each of these systems, we give sufficient descriptions of their protocols and explain the bugs we found and how they are discovered. We also summarize the benefits from WiDS Checker at the end of each case study. We will discuss relevant performance numbers and conclude with lessons that we have learned.

Application	# of lines	# of bugs	Lines of script
Paxos	588	2	29
Lock server	2,439	2	33
BitVault	17,582	3	181
Macedon-chord	2,468	5	86

Table 2: benchmark and result summary

4.1 Paxos

Paxos [13] provides a fault-tolerant distributed consensus service. It assumes an asynchronous communication model where messages can be lost and delayed arbitrarily. Processes can operate at arbitrary speed, may fail-stop and restart. Our implementation strictly follows the I/O-automata specification in [21], in which there are two types of processes: *leaders* and *agents*.[2] Leaders propose values to the agents round-by-round, one value for each round. Agents can accept or reject each received proposal independently, and a decision is made if the majority of the agents agree on the same round/value pair. The important safety property in Paxos is *agreement*: once a decision is made, the decided value will stay unchanged.

The main idea of the algorithm is to have leaders work cooperatively: endorse the latest accepted round/value pair in the system as they currently understand it, or propose their own if there is none. The following is an informal description. The protocol works in two phases, learning (steps 1 and 2) and submitting (steps 3 and 4):

1. A leader starts a new round by sending a *Collect* request to all agents with its round number n. The round number must be unique and increasing.

2. An agent responds to a *Collect* request with its latest accepted round/value pair (if any), if and only if the round number in the request is greater than any *Collect* requests to which it has already responded.

3. Once the leader has gathered responses from a majority of agents, it sends a *Begin* request to all agents to submit the value for round n. The submitted value is the previously accepted value in the highest-numbered round among the responses in Step 2, or the leader's own value if there is none.

4. An agent that receives a *Begin* message of round number n accepts the value (and accordingly updates its latest accepted value and round number), unless it has already responded to a *Collect* request with a higher round number than n.

```
declare_table Agent from PaxosAgent
column LastRound as m_last_accepted_round keep_version(2)
column LastValue as m_last_accepted_value
end_declare

declare_derived decision_value keep_version(2)
begin_python
   rnd_cnt = {};          # map from round number to count
   rnd_val = {};          # map from round number to submitted value
   for x in Agents :      # endorse an existing decision
      if x.LastRound in rnd_cnt.keys() :
            rnd_cnt[x.LastRound] += 1;
      else :              # record a new decision
            rnd_cnt[x.LastRound] = 1;
            rnd_val[x.LastRound] = x.LastValue;
   for round, count in rnd_cnt.items() :       # report the decision
      if count >= 4 : return rnd_val[round]
   return "NIL"
end_python
                          # decision shall never change
predicate GlobalConsistency{
   decision_value.version(-1) == decision_value.version(0)
   or decision_value.version(-1) = "NIL"
}
               # accepted round number should always increase
predicate AcceptedRoundIncreasing {
   (forall x in Agents)
   x.LastRound.version(-1) <= x.LastRound.version(0)
}
```

Figure 4: Checker script for Paxos

In our implementation, leaders choose monotonically increasing round numbers from disjoint sets of integers. An agent broadcasts to all leaders when it accepts a value so that leaders can learn the decision. Each leader sleeps for a random period before starting a round, until it learns a decision is made. The test was carried out using simulation with seven processes acting as both leaders and agents. To simulate message loss, we randomly dropped messages with a probability of 0.3.

We wrote two predicates that are directly translated from safety properties in the specification (see Figure 4). The Python snippet of decision_value calculates the value accepted by the majority of agents. The first predicate, *GlobalConsistency*, specifies the agreement property: all decision values should be identical. It is a distributed property derived from all agents. The second predicate, *AcceptedRoundIncreasing*, states a local property in the specification that the newly accepted round number increases monotonically in an agent.

The checker caught an implementation bug in Step 3 with the *GlobalConsistency* predicate, finding that after an agent accepted a new *Begin* message, the decision value changed to a different one. The root cause is that, in Step 3, after the leader gathers responses from a majority of agents, it sends a *Begin* request with the submitted value from the *latest* received agent response, instead of the *highest-numbered* round among all responses. The predicate breaks immediately after the *Begin* message is handled that changes the decision. Tracing back one single step in the message flow and replaying the code for Step 3 allows us to immediately identify the root cause.

The second bug is far more subtle. We found that with very small probability, the accepted round number in an agent may decrease, which violates the second predicate. We ran our test several hundred times, each of them having thousands of messages, but only caught one violation. Using replay to trace back from the violation point, we identified that the bug was not in the implementation, but in the specification itself. In Step 3 the *Begin* requests are sent to all agents; under message loss it is possible for an agent to receive a *Begin* request without the pairing *Collect* request of the round. This means that the agent can have its accepted round number greater than its round number responding to the *Collect* request (the two are kept in separate counters). Thus, based on Step 2 the agent may in the future respond to (and in Step 4 accept a value from) some smaller-numbered rounds, decreasing the accepted round number. With a small probability, the bug can actually break the ultimate agreement property. (We have constructed such a sequence of events starting with our violating trace). However, this is a corner case and the protocol is robust enough that it never happened in our test. After researching the specification carefully, we also understood where the original proof went wrong. This bug and our fix for the specification is confirmed by one of the authors.

Study on effectiveness. The checker gives precise bug points from thousands of events in both bug cases. After that, identifying the root cause becomes simply tracing back one or a few messages in replay. Replay (or only logging, if we dump states of Paxos nodes to the log) is necessary for understanding root causes, however, as *GlobalConsistency* is a distributed property that cannot be directly verified from logs. Without the checker a developer has to go through a huge number of events in replay trace and check the correctness property manually, which is very inefficient. The second predicate, though a local one, proves the usefulness of rigorous predicate checking for distributed systems. Without this predicate, we would miss the specification bug altogether.

4.2 Lock Server in Boxwood

The Boxwood [19] storage system implements a fault-tolerant distributed lock service that allows clients to acquire multiple-reader single-writer locks. A lock can be in one of the states: *unlocked, shared, exclusive*, and multiple threads from one or more clients acquire or release locks by communicating to a server. The server uses a *pendingqueue* to record locks for which transactions are still in flight.

The system was written in C#. In order to check it, we ported it to C++ and the WiDS APIs in about two weeks; most of the effort was spent on the language difference (e.g., the pointers). In the resulting code, almost every line can be mapped to the original code. This enables us to map bugs found by WiDS Checker to the original C# code. At first we checked the safety property that if multiple clients simultaneously own the same lock, they must be all in shared mode. However, during the experiment we did not find any violations in both simulated and reproduced runs. Next, we focused on deadlock and livelock checking on a larger scale and found both of them.

```
declare_derived pendingsize keep_version(2)
begin_python
  return Server[0].pendingsize
end_python

declare_derived last_change_time
begin_python
   if (pendingsize.version(0) != pendingsize.version(-1)) :
   return RuntimeInfo.current_time;
   return last_change_time;       # remain unchanged
end_python

declare_derived enough_time {
begin_python
  return  (Runtime.current_time – last_changed_time) / threshold
end_python

predicate PendingQueueIsEmpty auxiliary enough_time {
   pendingsize.version(0) == 0
}
```

Figure 5: Checker script for lock server

We ran the test inside a simulator and used 20 clients, one server and four locks. Each client has five threads that keep randomly requesting one of the locks and then releasing it after a random wait. Rather than writing sophisticated predicates to look for cycles of dependencies, we used a simple rule that if the protocol is deadlock-free, the *pendingqueue* should *eventually* be empty. The predicate is just that: return true if the *pendingqueue* is empty. Clearly there could be many false alarms. We attached an auxiliary output that computes how long the predicate remained broken since the last time the queue size changed (Figure 5).

Livelock. The livelock we discovered involves one lock and two clients. Based on replay and the message flow graph, we isolated the following bug scenario. Client A holds lock l in shared mode, and client B requests l in exclusive mode. The server inserts l to the $pendinglock$ queue and sends a revoke message to A. While this message is on its way, another thread in A requests to upgrade l to exclusive. According to the protocol, A needs to release l first before acquiring it again with the desired mode. A does so by first setting a $releasepending$ flag and then sending the server a release message.

According to the protocol, the server always denies a lock release request if a lock is in the $pendinglock$ queue. The revoke message arrives at A and spawns a revoking thread, which in turn was blocked because the lock is being released (i.e., the $releasepending$ flag is

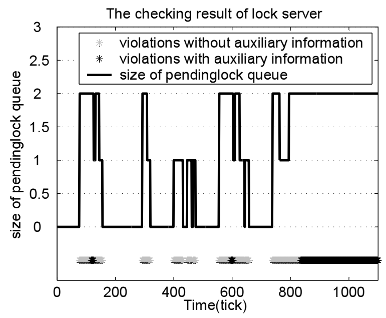


Figure 6: Violations reported in the livelock case. The stripe at the bottom contains all the violations; the dark ones are those with *enough_time* above the threshold.

set). When A finds its release is unsuccessful it resets the $releasepending$ flag and retries. However, the retry code is in a tight loop, and thus the release request is sent again and the $releasepending$ flag is set. As a result the revoke thread wakes up only to find that it is to be blocked again. This cycle continues onwards and repeats as a livelock. It is livelock in the sense that there is a small possibility that the blocked revoking thread can cut in after the release response is handled but before the next release retry occurs.

Figure 6 visualizes the violation reports in a run that discovered the livelock. There were altogether five rounds of competition and the bug appears at the final one. We use *enough_time* to screen out false alarm in violations. After screening, many false alarms are eliminated. Auxiliary output helped us to prioritize inspection of violations; otherwise the violations will be too large in number to inspect.

Deadlock. The deadlock case is more sophisticated. The client implementation has an optimization of using a socket pool for outstanding requests, and a thread will be blocked if it cannot find a socket to use. Because the socket pool is shared among threads, it creates extra dependencies and induces a deadlock.

We configured the socket pool to use only one socket. The resulting deadlock case involves four clients and two locks. The initial state is that A has shared lock l_1, and C has exclusive lock l_2. Next, B and D request exclusive mode on l_2 and l_1, respectively. After a convoluted sequence of events, including threads on A and B attempting to upgrade and release their locks, the system enters a deadlock state. Based on replay and message flow graph, we find the deadlock cycle, which consists of a lock acquiring thread whose request is blocked on the server because the lock is in the $pendinglock$ queue, a revoking thread blocked by the lock's $releasepending$ flag, and a release thread blocked by the unavailability of socket.

Study on effectiveness. Unlike the Paxos case in which the checker directly locates the bug point, here we do not have effective predicates to reveal deadlock/livelock; the predicate based on $pendingsize$ only provides hints for the starting point of the replay. So, the checker is more like an advanced "conditional breakpoint" on the replay trace, and mining the root cause heavily depends on the replay facility and the message flow visualization tool.

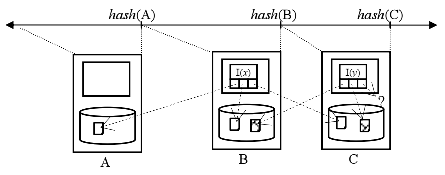


Figure 7: BitVault ID space and index structure. B and C are owners of object x and y, respectively. Object x has three replicas, whereas object y has one dangling pointer.

4.3 BitVault Storage System

BitVault [32] is a scalable and content-addressable storage system built by ourselves. The system is composed of more than 10 interdependent protocols, some of which incrementally developed over a stretch of two years. BitVault achieves high reliability with object replication and fast parallel repair for lost replicas. Self-managing routines in BitVault ensure that eventually every existing object in the system has its specified replication degree.

To understand the bugs, it is necessary to describe BitVault's internal indexing structure and repair protocols. BitVault uses a DHT to index the replicas of each object. Each object has a unique 160bit hash ID, and the entire ID space is uniformly partitioned by the hashes of the participating nodes, each node owning a particular ID range. An object's replicas can reside on any node, while its index (which records locations of the object's replicas) is placed on the *owner* node that owns the ID of the object. The design of the index enables flexible control of replica placement. Figure 7 shows the index schema. Mapping between ID space and nodes is achieved by a decentralized weakly-consistent membership service, in which every node uses heart-beat messages to detect node failures within its ID neighborhood, and maintains a local membership view of all nodes in the system. According to the membership view, the owner of an index can detect replica loss and then trigger the creation of an another replica. A replica also republishes itself to the new owner node of the index when it detects the change of the owner node.

BitVault has the following correctness properties derived from its indexing scheme: 1) correct index ownership, i.e., for each object, eventually the owner node holds the index; 2) complete reference, i.e., when the system stabilizes, there should be neither dangling replica references nor orphan replicas; 3) correct replica degree, i.e., in a stabilized system every object has exactly $degree$ (say 3) replicas. Because a node's membership view is guaranteed to converge *eventually*, all these must be considered liveness properties. We use these properties to check BitVault, and associate them with a stabilization measure based on membership change events. All experiments are conducted over an 8-node configuration in a production run, and we found the three bugs with the checker. Due to space limitations, we only explain two of them.

Replica loss due to protocol races. BitVault passes intensive testing before we added a routine that balances load between nodes by moving replicas. After adding the load balancing routine, we observed an occasional replica loss from our monitor GUI. Before we developed WiDS Checker, we did not actually know the root cause because the bug case was buried in irrelevant log events.

The checker catches a violation of the replica degree predicate saying that an object's replica number should be no less than 3. From the violation point we trace back a few messages and quickly found how the replica number for this object goes from 3 to 2. BitVault has a "remove-extra-replicas" routine that periodically removes over-repaired replicas which are caused by transient failures and retries during replica repairing. The bug was caused by a race between the load balancing routine and the remove-extra-replica routine, where the load balancing routine copies the replica from node A to B and deletes the replica in source node A, and at the same time the remove-extra-replicas routine detects that there are 4 replicas in total and chooses to delete the one in B.

Mishandling of transient failures. Sophisticated predicates enable more rigorous and thorough checks to catch bugs. As an example, the predicate of reference correctness ("no dangling references nor orphan replicas") checks the matching between index entries and replicas, and it helps us to identify a subtle bug caused by transient failures, which may degrade reliability. We killed a process in a node and restarted it after about 5 seconds. The predicate remained violated after quite a long time, even when the auxiliary measure for stabilization was high. Then we refined the predicate to output the owner of orphan replicas, which turned out to be the node suffering the transient failure. The bug is caused by mishandling of transient failures. The churn of the failed node cannot be observed by other nodes because the churn duration is shorter than the failure detection timeout (15 seconds). As a result, other nodes will not repair replicas or re-publish the index to this failed node, whose memory-resident state (e.g., the index) has already been wiped out by the failure.

Study of effectiveness. Both bugs are non-deterministic, complicated and sensitive to the environment, while catching the bugs and understanding the root causes require detailed memory states. Deterministic replay is necessary because we cannot dump all the memory states in logs in production runs. Like the Paxos case, the predicates directly specify the complex guarantee for correctness. Although they are liveness properties, predicate checking is very useful to pinpoint the starting point for inspection in replay. Suppose that we only have replay but not checking facility. For the first bug where replica loss has been observed, screening traces and finding out the root cause is tedious, while possible. In contrast, the second bug is difficult to even detect. This is because, eventually the replicas and owner nodes will notice the data loss through a self-managing routine in BitVault and repair the loss. Thus, the delay in repair is undetected and will degrade reliability.

4.4 Macedon-Chord

From the latest version of the Macedon release (1.2.1-20050531) [1], we generated the WiDS-based implementation for three protocols, RandTree, Chord, and Pastry. Due to space limitations, we only report our findings of Macedon-Chord. The Macedon description files already have logics of initialization, joining and routing, and we wrote our drivers to add our testing scenarios. The test is carried out in the simulator: 10 nodes join the ring around the same time and then remove one node after stabilization. We use the predicate in Figure 2 to check that the ring is well-formed, and add another to check fingers to be pointing to correct destinations.

Interestingly, this simple test altogether revealed five bugs. Two bugs are not caught by the predicates - they are programming errors that crash the simulation (divide-by-zero and dereferences of uninitialized pointers). The remaining three bugs are protocol implementation bugs caught by the predicates.

The first one caused a broken ring after the node leaves, caught by the $RingConsistency$ predicate[3]. Using replay to trace the origin of the incorrect successor value from the violation point, we found that the field of the successor's successor is wrongly assigned with the hash of the corresponding node's IP address, instead of the IP address itself.

The remaining two bugs caught by finger predicates cause problems in finger structures. One does not properly update the finger table, making the fingers converge to their correct targets much later than it needs to take. The last one is a mishandled boundary case. Let f be the ID of the i^{th} finger and $f.curr$ denote its current value. $f.start$ records the ID that is 2^i away from this node's ID. If $f.start$ equals to $f.curr$, then the current finger is the perfect one. When a new candidate node with ID y is evaluated, a checking function $isinrange$ is called, and $f.curr$ is replaced with y if y falls between $[f.start, f.curr)$. $isinrange$ regards the special case $[f.start, f.start)$ as the entire ring and returns true for arbitray y. As a result, the perfect finger is replaced by any arbitrary y. In later rounds the perfect finger will make it back, and the process continues to oscillate. Macedon-Chord is later re-written with Mace. The author confirmed that the first four bugs were removed as a result of this exercise, yet the last one still remains in the new version.

Study of effectiveness. Structured overlay protocols are perfect examples of the complexity of distributed logic. Based on system properties on topology structures, checking overlay protocols could be very effective and productive. Sometimes the violation report is sufficient to infer the root cause and find the buggy code, e.g., the oscillation bug reveals the buggy logic for choosing finger node ID, without the need for the replay.

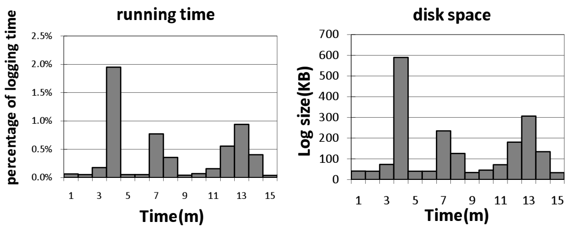


Figure 8: Logging overhead: (a) running time; (b) disk space.

Application	Original run	w/o checker	w/ checker
Paxos(simu.)	0.62	0.34	6.56
BitVault(deployed)	900.00	236.25	2219.77
Lock server(simu.)	5.34	3.14	11.99
Chord(simu.)	1.64	0.719	3.00

Table 3: Running time (in seconds) for evaluating predicates. Orignal run, w/o checker, and w/ checker columns show the running time for testing with log turned on, replay, and replay with predicate checking, respectively.

4.5 Performance and overhead

The logging overhead heavily depends on the applications. We performed a test run on BitVault with 4 nodes, each of which is a commodity workstation with 3GHz Pentium IV CPU and 512MByte memory, and the results are shown in Figure 8. The experiments consists of inserting 100 100KB objects at the 3^{rd} minute, crashing a node at the 6^{th} minute, rejoining it at the 9^{th}, and finally retrieving all objects at the 12^{th}. The replication degree is set to 3. The peak of the performance hit occurs on the 3^{rd} minute, with roughly a 2% runtime overhead. To collect logs more efficiently, we generate the objects that mostly contain a single value so as to achieve high compression rates. As a result, the log size reaches close to 600KB after compression. For uncompressed logs, the size is around 60MB.

Table 3 summarizes the performance of the checker. The second column shows how long it takes to perform the original run, with logging turned on. Depending on the message rate and complexity of predicate calculation, the checker slows down the replay between 4 and 20 times. The 15-minute BitVault run consumes about 37 minutes. We believe that given the benefits of using the checker, these overheads are acceptable for debugging tasks.

4.6 Discussions

Our experience validates a number of design points. In almost all cases, on-the-fly scripting has allowed us to adaptively refine predicates in response to predicate errors and to chase newly discovered bugs. This iterative process is especially useful when we check reproduced runs from deployed experiments, since the trace can be reused to find all the bugs it contains. The experiences also gave us a number of interesting lessons.

The advantage of predicate-based checking is its simplicity: it depends on only a snapshot of states (and sometimes augmented with a short history) to define invariant properties and catch violations. This methodology does not require the developer to build a state machine inside the checker that mirrors the steps of the implementation. At the same time, however, this means that we need to pay extra effort to trace the root cause backwards from violation point, which might be far behind. Time-travel with message flow definitely helps, yet it can be tedious if the violation involves long transactions and convoluted event sequences, as is the case of the Boxwood lock server. In these scenarios, the predicate checking is more like a programmable conditional breakpoint that people use to "query" the replay trace. Effectively pruning the events to only replay the relevant subset is one of the directions for our future work.

We also obtained considerable insight in terms of debugging distributed systems in general. The Paxos bug involves at least 5 processes, the deadlock case in the lock server involves 4 clients and 1 server, 2 locks, and the client has 6 flags in total. In both cases the error is deep and the system scale is larger than what a model checker is typically applicable to. It is therefore unclear whether a model checker can explore such corner states successfully. Also, 9 out of the 12 bugs have correct message communication pattern. For example, the bug found in the Paxos specification will not cause any violation of the contract between proposers and acceptors on sending and receiving messages. Thus, without dumping out detailed states, it is questionable whether log analysis is able to uncover those bugs.

5 Related Work

Our work is one of many proposals to deal with the challenging problem of debugging distributed systems. We contrast it with the three broad approaches below.

Deterministic replay. In response to the lack of control of testing in real deployment, building a time machine to enable deterministic replay is required [7, 8, 27]. Most of this work is for a single node. Our implementation is capable of reproducing the execution of a distributed system within one simulation process. Friday [9] enhances GDB with group replay and distributed watchpoints. We share their methodology that incremental refinement of predicates is an integral part of cyclic debugging process, however, replay in WiDS checker is much more efficient because we use one process to replay all instances. In addition to debugger extensions, WiDS checker provides a unified framework with advanced features tailored for distributed systems. It allows simulation with controlled failures, which is valuable for early development stages. It can trace user-defined object states with historical versions and evaluate predicates only at event-handler boundaries, and thus provides better functionalities and performance for predicate checking. These unique contributions of WiDS Checker are proven to be important for debugging. Our current drawback is that the tool is limited to applications written using the WiDS API or Macedon.

Model-check actual implementation. Our work complements these recent proposals [20, 30] that check actual implementations. Model checking explores the state space systematically, but the issue of state explosion typically restricts the scale. MaceMC [11] and WiDS checker share many design concepts, but differ fundamentally on how we test a system and hit bugs. MaceMC uses model checker with heuristics to deal with liveness properties, but has to tradeoff the scale of the system. As we discussed in Section 4.6, some bugs rely on a fairly large scale and low-level implementation details, and cannot manifest in a downscaled or abstracted system (e.g., the deadlock case in Boxwood derived from running out of the socket pool). For such cases, tools like WiDS checker which simulates low-level OS APIs and uses deployed testing with replay-based checking will be necessary.

Log-based debugging. Communication structures encode rich information. Logs can be used to identify performance bottlenecks, as advocated by Magpie [5], Pinpoint [6] and many others [4, 23]. The logs that WiDS Checker captures contain enough information to enable performance debugging, but the focus of this study is on correctness debugging. Pip [23] also proposes that log-based analysis can root out structural bugs. In general, we are more confident that log-based analysis can reveal performance bugs than structural ones (a close read on Pip's results seems to confirm this). As demonstrated by the bug cases in this study, a correct communication pattern is only the necessary but not the sufficient condition to enforce correct properties. Logging detailed states is prohibitively expensive; it is easier to log non-deterministic events and to reconstruct the states. From our experience, full-state inspection with time-travel is critical to identify the root cause of a correctness bug. A nice by-product is that it also enables us to exhaustively test all bugs in a given reproduced run, an endeavor that is usually quite labor-intensive. We also differ on how correctness is expressed. Pip can construct path expectations from history. Since we are taking logs, this alternative is also available. However, we believe that our assertions are more powerful. They represent the minimum understanding of a system's correct properties, and are much easier to write and refine than a model.

Singh et al. [26] propose to build an online monitoring and debugging infrastructure based on a declarative development language [16]. They require that the system is programmed with a highly abstracted deductive model so as to enable checking, while WiDS Checker mimics low-level OS API semantics (thread, sockets and file I/O) and enhance them with replay, in order to check predicates and find bugs in existing systems. In addition, their online checking methodology is restricted by the communication and computation overhead in distributed systems, and thus the checking facility is less powerful than the offline checking used in WiDS Checker. As online and offline checking complement each other, our future work is to look at interesting combinations of the two methods.

6 Conclusion and On-going Work

In a unified framework, WiDS Checker is capable of checking an implementation using both simulated as well as deterministically reproduced runs reconstructed from traces logged in deployment. Its versatile script language allows a developer to write and incrementally refine assertions on-the-fly to check properties that are otherwise impossible to check. Single console debugging and message-flow based time-travel allows us to quickly identify many non-trivial bugs in a suite of complex and real systems.

Our on-going work addresses the limitations in supporting legacy binaries with function interception techniques. We intercept OS system calls and APIs to change them into events, and use an event queue to schedule the execution, including thread switches, OS notifications, and socket operations. We also intercept a layer beneath the socket interface to add Lamport Clock annotation before network data chunks in a transparent way. By this means application can be logged and faithfully replayed in a transparent way, and we can futher bring the simulation, virtualization, and checking functions to legacy binaries in the fashion we performed with WiDS Checker.

7 Acknowledgments

We would like to thank our shepherd Petros Maniatis and the anonymous reviewers for their comments and suggestions. We are also indebted to Linchun Sun, Fangcheng Sun, Shuo Tang, Rui Guo for their help with the WiDS Checker experiments, as well as Roberto De Prisco, Lidong Zhou, Charles Killian, and Amin Vahdat for verifying the bugs we found.

References

[1] Macedon: http://macedon.ucsd.edu/release/.

[2] Phoenix compiler framework. http://research.microsoft.com/phoenix/phoenixrdk.aspx.

[3] WiDS release. http://research.microsoft.com/research/downloads/details/1c205d20-6589-40cb-892b-8656fc3da090/details.aspx.

[4] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *SOSP* (2003).

[5] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using magpie for request extraction and workload modelling. In *OSDI* (2004).

[6] CHEN, M., KICIMAN, E., FRATKIN, E., FOX, A., AND BREWER, E. Pinpoint: Problem determination in large, dynamic, internet services. In *Int. Conf. on Dependable Systems and Networks* (2002).

[7] DUNLAP, G. W., KING, S. T., CINAR, S., BASRAI, M. A., AND CHEN, P. M. Revirt: enabling intrusion analysis through virtual-machine logging and replay. *SIGOPS Oper. Syst. Rev. 36*, SI (2002).

[8] GEELS, D., ALTEKAR, G., SHENKER, S., AND STOICA, I. Replay debugging for distributed applications. In *USENIX* (2006).

[9] GEELS, D., ALTEKARZ, G., MANIATIS, P., ROSCOEY, T., AND STOICAZ, I. Friday: Global comprehension for distributed replay. In *NSDI* (2007).

[10] JUMP, M., AND MCKINLEY, K. S. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL* (2007).

[11] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Finding liveness bugs in systems code. In *NSDI* (2007).

[12] LAMPORT, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM 21*, 7 (1978).

[13] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (1998).

[14] LIN, S. D., PAN, A. M., GUO, R., AND ZHANG, Z. Simulating large-scale p2p systems with the wids toolkit. In *MASCOTS* (2005).

[15] LIN, S. D., PAN, A. M., ZHANG, Z., GUO, R., AND GUO, Z. Y. Wids: an intergrated toolkit for distributed system development. In *HotOS* (2003).

[16] LOO, B. T., CONDIE, T., HELLERSTEIN, J. M., MANIATIS, P., ROSCOE, T., AND STOICA, I. Implementing declarative overlays. *SIGOPS Oper. Syst. Rev. 39*, 5 (2005).

[17] LYNCH, N. *Distributed Algorithms*. 1996, ch. 8.

[18] LYNCH, N., AND TUTTLE, M. An introduction to input/output automata. In *Technical Memo MIT/LCS/TM-373* (1989).

[19] MACCORMICK, J., MURPHY, N., NAJORK, M., THEKKATH, C. A., AND ZHOU, L. Boxwood: Abstractions as the foundation for storage infrastructure. In *OSDI* (2004).

[20] MUSUVATHI, M., AND ENGLER, D. Model checking large network protocol implementations. In *NSDI* (2004).

[21] PRISCO, R. D., LAMPSON, B. W., AND LYNCH, N. A. Fundamental study revisiting the paxos algorithm. *Theoretical Computer. Science. 243*, 1-2 (2000).

[22] QIN, F., LU, S., AND ZHOU, Y. Safemem: Exploiting ecc-memory for detecting memory leaks and memory corruption during production runs. In *HPCA* (2005).

[23] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the unexpected in distributed systems. In *NSDI* (2006).

[24] RODRIGUEZ, A., KILLIAN, C., BHAT, S., KOSTIC, D., AND VAHDAT, A. Macedon: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI* (2004).

[25] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., AND ANDERSON, T. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst. 15*, 4 (1997).

[26] SINGH, A., ROSCOE, T., MANIATIS, P., AND DRUSCHEL, P. Using queries for distributed monitoring and forensics. In *EuroSys* (2006).

[27] SRINIVASAN, S. M., KANDULA, S., ANDREWS, C. R., AND ZHOU, Y. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX* (2004).

[28] STOICA, I., MORRIS, R., LIBEN-NOWELL, D., KARGER, D. R., KAASHOEK, M. F., DABEK, F., AND BALAKRISHNAN, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw. 11*, 1 (2003).

[29] YANG, H., PIUMATTI, M., AND SINGHAL, S. K. Internet scale testing of pnrp using wids network simulator. In *P2P Conference* (2006).

[30] YANG, J., TWOHEY, P., ENGLER, D., AND MUSUVATHI, M. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst. 24*, 4 (2006).

[31] YU, Y., RODEHEFFER, T., AND CHEN, W. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP* (2005).

[32] ZHANG, Z., LIAN, Q., LIN, S. D., CHEN, W., CHEN, Y., AND JIN, C. Bitvault: A highly reliable distributed data retention platform. In *MS Research Tech Report (MSR-TR-2005-179)* (2005).

Notes

[1]WiDS, a recursive acronym that stands for "WiDS implements Distributed System", is the name of the toolkit.

[2]For conciseness, we omit the learners of paxos in our description.

[3]This is only a simplified ring consistency predicate for illustration purpose. The more complicated one which also detects disjoint and loopy rings would possibly catch more problems.

X-Trace: A Pervasive Network Tracing Framework

Rodrigo Fonseca George Porter Randy H. Katz Scott Shenker Ion Stoica
Computer Science Division
Univ. of California, Berkeley
Berkeley, Calif. 94720-1776
{rfonseca,gporter,katz,shenker,istoica}@cs.berkeley.edu
http://xtrace.cs.berkeley.edu

Abstract

Modern Internet systems often combine different applications (e.g., DNS, web, and database), span different administrative domains, and function in the context of network mechanisms like tunnels, VPNs, NATs, and overlays. Diagnosing these complex systems is a daunting challenge. Although many diagnostic tools exist, they are typically designed for a specific layer (e.g., traceroute) or application, and there is currently no tool for reconstructing a comprehensive view of service behavior. In this paper we propose X-Trace, a tracing framework that provides such a comprehensive view for systems that adopt it. We have implemented X-Trace in several protocols and software systems, and we discuss how it works in three deployed scenarios: DNS resolution, a three-tiered photo-hosting website, and a service accessed through an overlay network.

1 Introduction

Internet services are built out of distributed components (e.g., load balancer, web server, backend database), make use of sophisticated network mechanisms (e.g., VPNs, NATs, overlays, tunnels), and can span multiple administrative domains (e.g., the client's web proxy and the server's load balancer). When these complex systems misbehave, it is often quite difficult to diagnose the source of the problem.

As an example, consider the infrastructure serving Wikipedia [27]. As of June 2006, they had servers spread across 3 sites, comprising 33 web caches chosen via DNS round-robin assignments, 4 load balancers, 105 web servers, and 14 database servers. A user's request transits a cache server, and may also transit a load balancer, a web server, and a database. Caching is done at each of these levels. Now suppose a user updates a page on Wikipedia, and fails to see her updates upon reloading the page. It is difficult to identify which cache, at which level, is returning stale data. Even if logs are kept, there is no common mechanism to determine which logs to examine, or to correlate entries across multiple logs. It may also not be possible for administrators to reproduce the problem, since their requests would most likely take a different path through the system.

Diagnostic tools do exist, but many of them are limited to a particular protocol. For instance, `traceroute` is useful for locating IP connectivity problems, but can't reveal proxy or DNS failures. Similarly, there are numerous alarm and monitoring suites for HTTP, but they cannot diagnose routing problems. While these tools are undoubtedly useful, they are also typically unable to diagnose subtle interactions between protocols or provide a comprehensive view of the system's behavior.

To this end, we have developed an integrated tracing framework called X-Trace. A user or operator invokes X-Trace when initiating an application task (e.g., a web request), by inserting X-Trace metadata with a task identifier in the resulting request. This metadata is then propagated down to lower layers through protocol interfaces (which may need to be modified to carry X-Trace metadata), and also along all recursive requests that result from the original task. This is what makes X-Trace comprehensive; it tags all network operations resulting from a particular task with the same task identifier. We call the set of network operations connected with an initial task the resulting *task tree*.

Constructing the task tree requires understanding the causal paths in network protocols. While in general this may be difficult, in most of the situations we have considered so far this is relatively straightforward: for example, a recursive DNS query is clearly causally linked to the incoming request. X-Trace requires that network protocols be modified to propagate the X-Trace metadata into all actions causally related to the original task. This involves both understanding calls to lower-level protocols (e.g., HTTP invoking TCP) and initiating forwarded or recursive requests.

X-Trace-enabled devices log the relevant information connected with each tagged network operation, which can then be reported back. The trace information associated with a task tree gives the user or operator a comprehensive view of what network operations were executed as part of a task. To illustrate, Figure 1 shows an example of the task tree involved in a simple HTTP request

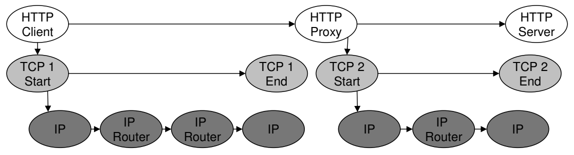


Figure 1: A proxied HTTP request and the logical causal relations among network elements visited.

through a proxy, showing the causal relations between operations in the HTTP, TCP, and IP layers. X-Trace task trees are runtime traces of a task execution, and so long as individual components are integrated into the framework, there is no need for prior configuration of their dependencies.

Diagnosing problems often requires tracing a task across different administrative domains (which we will call ADs). ADs may not wish to reveal internal information to each other, or to end users. Accordingly, X-Trace incorporates a clean separation between the client (user or operator) that invokes X-Trace, and the recipient of the trace information. For instance, when an end user notices a problem and invokes X-Trace, the trace information from her home network is delivered to her locally, the trace information from her ISP is delivered to the ISP support center, and the trace information from the web site she was accessing is sent to the web site operator. Each of these parties can then deal with the information as they see fit; sharing it with others, keeping it private, or even not collecting it at all. The fact that X-Trace gives them a common identifier for the task enables them to cooperate effectively if they so choose.

Realistically, we know all layers in the stack and different ADs will not deploy X-Trace-enabled protocols and devices simultaneously. However, individual protocols, applications, or ADs can benefit immediately from X-Trace if they support it. If a particular protocol or application gets instrumented alone, one gets horizontal slices of the task tree, which are useful for developers and users. If an AD alone deploys it on multiple layers within its network, it gets to internally visualize the portion of the task tree that happened inside of its domain. In addition, there is a "network effect" for adoption: as more protocols and networks integrate into the framework, X-Trace offers a common framework for their sharing of this information, increasing the value for all parties.

There has been much prior work on the study of application behavior, network monitoring, and request tracking. We discuss this related work in detail in Section 7 and only note here that the main differentiating aspect of X-Trace is its focus on tracing multiple applications, at different network layers, and across administrative boundaries. Section 4 highlights these features in the context of three specific examples. However, X-Trace is applicable to a wide variety of other protocols, such as SIP, RPC, and email.

While we feel that X-Trace provides a valuable service, it certainly has significant limitations. They are discussed in detail in Section 6, but we note them briefly here. First, implementing X-Trace requires modifications to clients, servers, and network devices; protocols that can't already do so must be altered to carry X-Trace metadata, and their implementations must log the relevant trace information. While these changes are conceptually simple, in practice retrofitting X-Trace into existing applications is a process of varying difficulty; our experiences in this regard ranged from trivial to quite challenging. Second, when X-Trace is only partially deployed, the ability to trace those parts of the network is impaired, sometimes entirely. Third, lost trace reports can limit reconstruction of the request tree and can lead to false positives in diagnosing faults (i.e., the lack of trace data may be interpreted as a failure). Fourth, our enforcing a tree structure on the set of network operations related to a particular task means that there are some request topologies that we cannot capture. For example, X-Trace is not able to naturally capture requests that rendezvous at a node where they are merged into a single request. It isn't clear, for instance, what should be considered causally-related in a routing protocol.

Because X-Trace only records paths that were taken, it is not a tool to assert global invariants about all possible paths. There are many problems for which X-Trace will not determine the cause, but will rather show the effect. While not an introspective debugger, it will point out the components involved in the operation, guiding the use of other tools to verify the cause. Examples of these cases are state corruptions that would cause a router to misroute packets, or an overloaded CPU that would cause a message to be dropped.

The rest of the paper is structured as follows. In Section 2, we describe the model and architecture of X-Trace. In Section 3, we describe our implementation of the X-Trace architecture. Section 4 describes three deployments of X-Trace and pinpoint six network error conditions. Section 5 discusses other uses of the system. In Section 6, we discuss the limitations of and security considerations raised by X-Trace. In Section 7, we discuss at length how X-Trace relates to previous work. Lastly, we conclude in Section 8.

2 Design Principles and Architecture

2.1 Design Principles

A key function of X-Trace is to reconstruct the task tree of all sub-operations making up the task. We now consider three principles that guided our design:

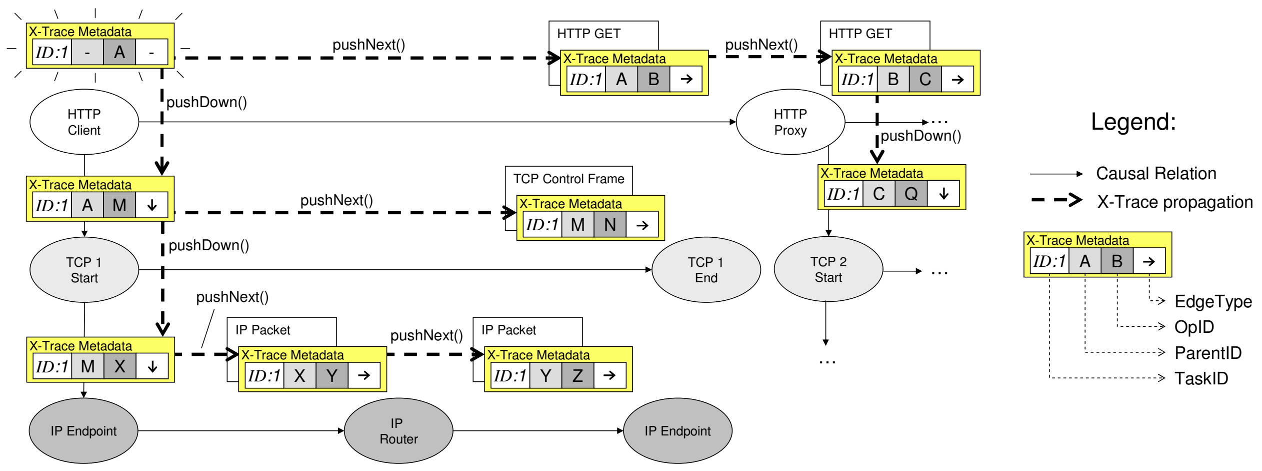


Figure 2: Propagation of X-Trace metadata in the example in Figure 1. Causal edges in the task tree are captured by the *ParentID*, *OpID*, and *EdgeType* fields. The *TaskID* remains the same for all the operations that comprise the task.

1. The trace request should be sent in-band, rather than in a separate probe message.

The first principle highlights our desire to probe what happens on the actual datapath we want to diagnose. Out-of-band probes might not end up following the same path as the original datapath. It follows that we need to *add metadata to the same datapath that we want to trace*. In X-Trace this metadata contains an identifier common to all operations in a task tree, which is added to messages and propagated by devices along the entire path.

2. The collected trace data should be sent out-of-band, decoupled from the original datapath.

This principle relates to gathering of trace information. If we appended trace information to the metadata encoded in the datapath, then we might lose this information in the event of network failure. Also, this would increase the overhead of messages. Obtaining trace data during periods of failure is especially important to this work. It follows that *we need an out-of-band, orthogonal mechanism to record and collect trace data*. Additionally, by decoupling trace reporting from the datapath, we lessen the impact of X-Trace on the datapath's latency.

3. The entity that requests tracing is decoupled from the entity that receives the trace reports.

As we discuss in §2.3 below, separating the user who inserts the X-Trace metadata in the datapath from the destination of the trace reports generated by components along the path allows for flexible disclosure policies of the trace information for each AD. Each AD keeps control of the information, while the common identifier allows them to cooperate in solving problems if necessary.

X-Trace places the *minimal necessary mechanism* within the network, while still providing enough information to reconstruct the path. The data itself is not kept in the network path, but rather reported to specific places determined by ADs. The X-Trace metadata contains enough information for ADs to communicate trace information back to the user if it so chooses.

2.2 X-Trace Metadata

In the following section, we describe the format and structure of the tracing metadata introduced by our system, as well as the way that metadata is propagated through applications.

Format and structure X-Trace metadata is the information placed into each layer to support the X-Trace framework. It is inserted into a network task by the client, if it is X-Trace capable. For legacy clients, devices in the network can add them. Network operators can insert X-Trace metadata for operations traversing their AD.

Within that metadata is a task identifier, which uniquely identifies each network task. This identifier should be unique among all of the reports accessed by an X-Trace user. X-Trace metadata is carried by the extension, option, or annotation fields within each network protocol. Examples of such fields are IP options, TCP options, and HTTP headers. It is replicated across layers, ensuring that devices on the path can access it without having to violate layering.

The metadata contains an optional field, *TreeInfo*, used by nodes to record causal relations between operations. It consists of a three-tuple: (*ParentID*, *OpID*, *EdgeType*). *ParentID* and *OpID* encode edges in the task tree. The *EdgeType* field indicates the type of that edge: either connecting two adjacent nodes at the same layer, or between a node at one layer with a node at a lower layer. The *ParentID* and *OpID* fields should be unique with respect to one task identifier. We describe how network devices manipulate these fields below. An optional destination field is used to send trace data to interested parties. This is described in detail in below, and its security implications are addressed in Section 6.

Figure 2 shows in full detail the contents and the propagation of X-Trace metadata (described in the next section) in part of the task tree from Figure 1. In particular, the successive values of the *ParentID*, *OpID*, and *EdgeType* fields allow the complete task tree to be reconstructed for this *TaskID*.

Propagation: `pushDown()` **and** `pushNext()` Devices and network elements on the path are responsible for propagating the X-Trace metadata along the path using two simple primitives: `pushDown()` and `pushNext()`. These primitives have the goal of ensuring that X-Trace metadata stays with the datapath. They manipulate the *TreeInfo* field of the X-Trace metadata, as shown in Table 1, recording the causal relations between operations in the path. The table shows how the fields in the *current* X-Trace metadata are mapped into the *next* metadata, for both primitives. The `unique()` function returns an identifier that is unique in the context of one *TaskID*.

The `pushDown()` primitive is responsible for copying X-Trace metadata from one layer to the layer below it. In Figure 2, all of the vertical arrows represent `pushDown()` operations. In this case, the HTTP proxy has to call `pushDown()` to copy the metadata into the newly generated TCP 2 connection. Likewise, the TCP process in the proxy has to call `pushDown()` to copy this metadata down to the new IP path. Note that we do not make any a priori assumptions as to the number or ordering of layers in a protocol exchange: `pushDown()` works recursively, with each layer only naturally interacting with the layer immediately below.

`pushNext()` is a primitive used by nodes on the datapath to propagate X-Trace metadata to the next hop in the same layer. In Figure 2, the HTTP proxy creates a new HTTP connection to the server. It calls `pushNext()`, which copies the metadata into the headers of that new connection, and captures the causal link between the two. All horizontal edges in the figure are `pushNext()` operations at their respective layers.

Since the X-Trace metadata is embedded into the messages at each layer, propagation happens at the same time as the messages are sent. In particular, if messages are stored for later forwarding, as is the case with email messages [10], the causal relations will still be preserved and recorded properly.

2.3 Task Tree Reconstruction

Collecting trace data with reports When a node sees X-Trace metadata in a message at its particular layer, it generates a report, which is later used to reconstruct the datapath. This report generation operation is separate from propagating X-Trace metadata, and is specific to the tree reconstruction aspect of our application.

TreeInfo operations	
`pushNext()`	next.parentID $\Leftarrow$ current.opID next.opID $\Leftarrow$ unique() next.type $\Leftarrow$ NEXT
`pushDown()`	next.parentID $\Leftarrow$ current.opID next.opID $\Leftarrow$ unique() next.type $\Leftarrow$ DOWN

Table 1: Effect of both propagation primitives mapping a *current* X-Trace metadata to a *next* metadata. `unique()` returns an ID unique within one task.

Reports contain a local timestamp, the *TaskID* they refer to, and information that is specific to the node sending the report. Devices only report information accessible at their own network layer. For example, an HTTP cache may report on the URI and cookie of the request, and the action taken upon receiving the request. It can also add systems information such as the server load at the time. IP routers, on the other hand, report information contained in the IP headers of packets, such as source and destination addresses, and can add other relevant performance information such as current queue lengths.

The reports generated by devices within one AD are kept under the control of that AD, according to its policy. That policy could be to store all the reports in local storage, such as a database. The AD can use this store of reports to diagnose and analyze flows transiting its network. Section 4.2 shows how a web hosting site uses locally generated and stored reports to diagnose faults in its components.

The X-Trace metadata has an optional *Destination* field. If present, this field signals that a user (located at that destination) is interested in receiving the trace data as well. This user might be the client, or it could be any delegated report server. This indirection is useful for users behind NATs, since they are not addressable from the Internet. The AD uses its policy to respond to this request. The simplest policy is for each device to just send reports directly to the indicated destination, which would collect them and reconstruct the task tree. This may not be desirable, though, because AD's in general will want to control who has access to what granularity of data. One possible mechanism that uses indirection works as follows. The AD still collects all reports locally in a private database. It then sends a special report to the user, containing a pointer to the report data. The pointer could be the URL of a page containing the trace data. This gives each AD control of the visibility of the trace information, by requiring users authenticate

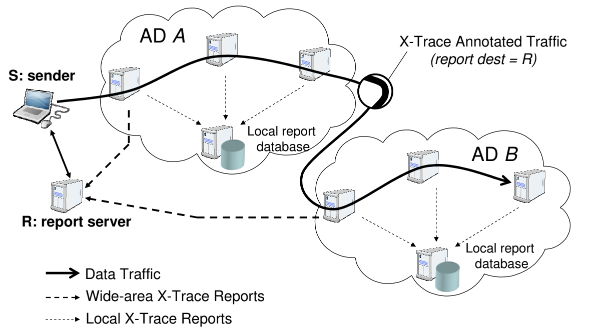


Figure 3: An example of wide-area reporting. The client embeds X-Trace metadata with a message, setting the report destination to R. Different ISPs collect reports locally, and send pointers to R so that the client can later request the detailed reports.

themselves when they fetch the data. The AD can make use of this authentication information when choosing the level of detail of the report information returned to the user. We describe this usage in more detail in Section 3. Note that all the information needed to get a report to a user is kept in the X-Trace metadata, meaning that nodes in the network do not need to keep any per-flow state to issue reports.

Figure 3 shows a sender S who sets the destination for reports as being the report server R. ADs A and B send pointer reports to R, and either the client or R itself fetches these reports later. A special case is when the *user* of X-Trace is in the same AD as the devices generating reports, such as network operators performing internal troubleshooting. X-Trace metadata gets added at the AD ingress points. The network operators go directly to the local report databases, and there is no need to use the destination field in the metadata.

Offline reconstruction of the task tree Task tree reconstruction is an offline process performed by the user that reconstructs the request path of the data connection. After the user collects reports from the reporting infrastructure, they examine them to reconstitute the request tree. Each of the reports is treated as a directed edge, either a "down" edge or a "next" edge, corresponding to `pushDown()` and `pushNext()` operation. After reconstructing the tree, the client can examine the nodes and paths that the request took. For transitory errors, this tree serves as a permanent record of the conditions that existed at the time of the connection. Additionally, any performance data included by the devices in the reports can be used to correlate failures in the datapath with devices that may be under-performing due to overload.

The reconstructed tree is the end product of the tracing process, and can be stored, associated with trouble ticket systems, or used by operators as a record of individual failure events for reliability engineering programs.

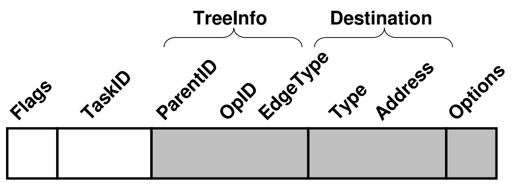


Figure 4: The X-Trace metadata and its fields. Shaded fields are optional.

3 Implementation

In this section we describe how we implemented the architecture described above. We discuss the representation of the X-Trace metadata and its propagation, a local reporting infrastructure, and a prototype for inter-AD reporting, as well as a simple procedure to reconstruct a task tree from a series of reports. We present some microbenchmarks, and close the section with a discussion of issues that arise when integrating X-Trace into new and existing protocols and applications.

3.1 Identifier format and semantics

Figure 4 shows the format with which we encode the X-Trace metadata. It consists of two required fields, and three optional ones:

Flags: The flags field contains bits that specify which of the three optional components of the X-Trace metadata are present: *TreeInfo*, *Destination*, and *Options*.

TaskID: Our design supports 4, 8, 12, or 20 byte integer fields to represent the *TaskID*. The *TaskID* must be unique within 1) a window of time, and 2) a reporting domain. The window of time must be long enough so that no two tasks that overlap in time share the same ID.

TreeInfo: *(Optional)* The *TreeInfo* field holds three subfields: *ParentID*, *OpID*, and *EdgeType*. *ParentID* and *OpID* are each 4 bytes long. These IDs must be unique within the context of a single task ID. We implement the `unique()` function as a random number generator. The *EdgeType* field is implemented as one byte, with only two values currently encoded: NEXT and DOWN.

Destination: *(Optional)* X-Trace metadata can optionally contain a destination, which is a network address that X-Trace reports should be sent to, as described in Section 2.3. The *Destination* field consists of two portions, a *type*, and an *address*. Currently implemented *type*s are shown in Table 2.

Options: *(Optional)* To accommodate future extensions to the X-Trace identifier format, we include an options mechanism. The Options block, if present, consists

Type	Protocol	Destination
Explicit	UDP	IPv4:port
	TCP	IPv4:port
	I3	I3 id
	XMLRPC	OpenDHT key
Implicit	Local	Configured
	Proxy	Configured

Table 2: Types of X-Trace report destinations.

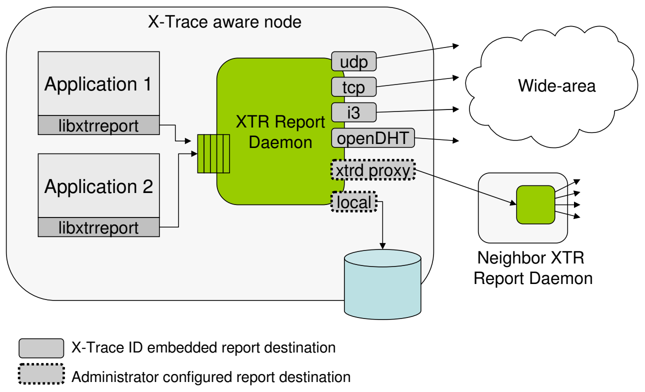


Figure 5: X-Trace reporting architecture.

of one or more individual options. Each consists of a type, a length, and then a variable length payload.

3.2 Reporting infrastructure

Report format A report is an ASCII message consisting of a header section followed by a body section. The first line of the header identifies the layer issuing the report. The rest of the headers are specified as key-value pairs, similar to headers in RFC 822 [10]. The body of the report is free-form, and the content is set by the device issuing the report and other operator policy.

Reporting libraries and agents Included with X-Trace is `libxtrreport`, a reference implementation of a client library that can be linked into applications for issuing reports. This library is very thin, and simply relays reports to a locally running daemon process.

The report daemon (see Figure 5) uses a UDP socket to listen for reports from the `libxtrreport` library. A thread listens for these reports, and places them on a queue. Another thread pulls reports off this queue, and sends them to the appropriate handler module. These modules, which run in separate threads, can forward the report to another report server, send it to a service like OpenDHT [21], or send it to any of the other destinations listed in Table 2. For local destinations, we make use of a Postgres SQL database for report storage.

We also implemented a packet sniffing application that can send reports on behalf of services and applications that cannot be modified to include `libxtrreport`. This application snoops network traffic using the `libpcap` library, sending reports for any protocols that it supports. Currently, this application supports the IP and TCP protocols. Network switches can make use of port mirroring to mirror traffic to this agent.

Inter-AS reporting We implemented a special case of Inter-AS reporting in the web hosting scenario described in Section 4.2. The front end webservers included two HTTP headers in the response sent back to the client. The first contains a URL for collecting trace information about the request. The second is the X-Trace task identifier associated with the network operation. This is included to simplify handling at the client, as well as for environments in which the X-Trace metadata was added by the frontend webservers. We wrote a Firefox extension that reads these HTTP headers, and provides the user with a visual indicator that the page they are visiting is “X-Trace enabled”, as well as a button they can click to fetch the trace data from the provided URL.

3.3 Offline tree reconstruction

Our implementation of the task tree reconstruction is quite simple, and can serve as the foundation for other, more complex, visualizations. We initially start by building a graph, G, consisting of the node represented by the first report. For each additional report we receive, we look for its parent (given by its ID_{parent} field) in the tree. If this new node’s edge type is NEXT, we attach the node on the same level as the parent. If the node type is DOWN, we attach the node at the level below the parent.

3.4 Performance

We tested the performance of the metadata propagation and the reporting aspects of our reference implementation of X-Trace. For the propagation, we measured the latency of `pushNext()`. This operation is blocking, and if implemented in a router, would have to be performed on a per-packet basis on the forwarding path.

We implemented `pushNext()` in the C language and tested it on a 64-bit Intel Pentium 4 CPU running at 3.2 GHz. We applied `pushNext()` to 576-byte packets, and found the average time required to be $0.71\mu s$. Taken in isolation, this processor could apply the operation to over 1.4 million packets per second. In fact, hardware implementations could be much faster.

To test the performance of the reporting infrastructure, we used the Apache web benchmarking tool, `ab`, against two otherwise identical Apache websites: one with reporting turned on and one without. The report store in this test was a separate Postgres database. Of the 10,000 requests we issued to the site, none of the reports were dropped by the reporting infrastructure. The regular server sustained 764 requests/sec, with a mean latency of 1.309 ms. The X-Trace enabled server sustained 647 requests/sec, with mean latency of 1.544 ms, which shows a 15% decrease in total system throughput.

3.5 Providing Support for X-Trace

Adding X-Trace support to protocols and applications involves three steps: (i) adding X-Trace metadata to messages exchanged, (ii) adding logic to propagate the X-Trace metadata within the implementation, following causal paths, and (iii) optionally adding calls to generate reports at interesting points of the message flow.

(i) Metadata To support X-Trace, a layer or application must embed X-Trace metadata in the messages it exchanges with peers. The difficulty of this for existing protocols depends on their specification. For example, it is simple for HTTP, because its specification [11] allows for extension headers, and dictates that unknown extensions be forwarded unmodified to next hops by proxies. Other protocols like SIP [22], e-mail [10], IP, TCP, and I3 share this characteristic. For protocols without an extension mechanism, one has to resort to either changing the protocol or overloading some existing functionality. In the implementation of Chord that comes with I3 we had to create a new type of message. Table 3 gives details on adding metadata to these and some other protocols.

(ii) Propagation Applications must support two aspects of X-Trace identifier propagation: (a) carrying X-Trace metadata between incoming and outgoing messages, and (b) manipulating the metadata with `pushDown()` and `pushNext()` operations to correctly record the causal relations. We implemented support in C/C++, Java, and PHP for easily manipulating X-Trace metadata, including performing the `pushDown()` and `pushNext()`, such that few lines of code need to be added to perform (b), once (a) is in place.

In our experience, we found that capturing the causal connections within the application presented the highest variability in difficulty, as it requires understanding how received messages relate to outgoing messages, and may require following long chains of calls within the implementation. If the implementation associates a context data structure with the processing of a message, it may be easy to add X-Trace metadata to the data type, which gets carried with the processing flow automatically. Apache and I3 fall into this category. Other implementation structures require more work, as in the case of Chord: we had to create a parallel path of functions with an extra X-Trace metadata parameter following the call path from receiving the message until sending it. Instrumenting concurrency libraries and runtime environments may ease or automate this propagation [9, 20, 8]. We are currently adding X-Trace support for libasync [17].

```
Original Forwarding Code
forwardMessage(msg)
    dest = nextHop(msg)
    lowerLayer.send(msg,dest)

With added X-Trace Propagation
forwardMessage(msg)
    dest = nextHop(msg)
    xtr = msg.getXTraceMetadata()
    /* Propagate to the next hop */
    msg.setXTraceMetadata(xtr.pushNext())
    /* Propagate to the lower layer */
    lowerLayer.setXTraceMetadata(xtr.pushDown())
    lowerLayer.send(msg,dest)
```

Figure 6: Pseudo-code highlighting changes for capturing causal relations with X-Trace

The pseudo-code shown in Figure 6 shows a typical example of the calls that are needed for full identifier propagation in the forwarding function of an application. We assume that the message abstract data type provides methods for getting and setting X-Trace metadata in the message, and that the lower layer also provides an API to set the X-Trace metadata for its messages. This example is patterned on the code we implemented in the applications and protocols we modified.

(iii) Integrating Reporting The remaining task is to get network elements – devices, protocol stacks, and applications – to issue reports of interesting events. For hardware devices like routers and appliances, one needs to modify the software running on the control processor. However, using the feature of port mirroring in switches, a network administrator can insert nodes that would report on traffic seen without slowing down the data path. The routers would still have to do the propagation, but not bother to call reporting functions. For software implementations, it is straightforward to integrate the reporting library, which is similar to adding a logging subsystem to the application.

4 X-Trace Usage Scenarios

In this section, we describe several scenarios where X-Trace could be used to help identify faults. We discuss three examples in detail–a simple web request and accompanying recursive DNS queries, a web hosting site, and an overlay network. We deployed these examples within one AD, and thus do not make use of the wide-area reporting mechanism. We follow these examples with a description of other scenarios.

Table 3: Support for adding metadata to some protocols. We have implementations for the protocols in *italics*.

Protocol	Metadata	Comment
HTTP, SIP, Email	Extension Header	Out-of-the box support for propagation. The only change is for causal relations.
IP	IP Option	Automatic propagation. Dropped by some ASs, wide-area support varies [12].
TCP	TCP Option	One-hop protocol, no next hop propagation. Linux kernel changes are needed.
I3	I3 Option	Support for options, but had to add handling code.
Chord[a]	No support	Mirrored augmented call path for new X-Trace data message.
DNS	EDNS0 OPT-RR	The EDNS0 [26] extension to DNS allows metadata to be added to messages.
SQL	SQL Comment	Possible to encode X-Trace metatada within a SQL comment.
UDP, Ethernet	No support	Must change protocol or use shim layer.

[a]The Chord implementation bundled with the I3 distribution.

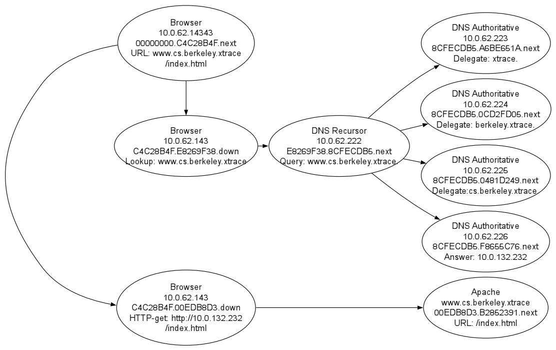


Figure 7: The complete HTTP and recursive DNS tree recovered by the X-Trace tool

4.1 Web request and recursive DNS queries

Overview The first scenario that we consider is that of requesting a web page from a server. Figure 7 shows the tree corresponding to a simple web request. The user starts by typing a URL into her browser, in this case *http://www.cs.berkeley.xtrace/index.html*. The browser's host first looks up the provided hostname using a nearby DNS resolver, which returns the IP address of that host (10.0.132.232). If the resolver does not have the requested address in its cache, it will recursively contact other DNS servers until a match is found. It can then issue the HTTP request to the resolved IP address.

Tracing each of these "subtasks" is a challenge: HTTP requests could be forwarded through proxies or caches, masking their ultimate destination. DNS requests are recursive in nature, are cached at intermediate servers, and span different administrative domains. This can easily lead to misconfigurations and inconsistent views.

X-Trace support We added support for X-Trace to the DNS protocol by using the EDNS0 [26] extension mechanism. This backwards-compatible mechanism allows metadata to be associated with DNS messages, and is increasingly supported in the wide area. We modified a DNS client library, an authoritative DNS server, as well as a recursive DNS resolver to support X-Trace metadata propagation and reporting.

We deployed this software in our local testbed, and created a parallel top-level domain (.xtrace). Figure 7 shows the final tree. In this example, the task has two subtasks, indicated by `pushDown()`: resolving the name, and fetching the page. A Java-based web browser issues the query to the DNS client library, which encapsulates the X-Trace metadata (after calling `pushNext()`) in an EDNS0 field of the query. This query is forwarded to the resolver on 10.0.62.222, which recursively looks up the address in other, authoritative nameservers, after calling `pushNext()` at each step of the recursion. Lastly, each of our authoritative nameservers issues reports when they receive queries with X-Trace/EDNS0 records in them. When the name resolution is complete, the browser issues an X-Trace enabled HTTP query (after calling `pushNext()`).

Fault isolation An X-Trace enabled DNS might uncover several faults that are difficult to diagnose today. At each step of the recursive resolution described above, servers cache entries to reduce load on the top-level servers. A misconfigured or buggy nameserver might cache these entries longer than it should. If a server's IP address changes, these out-of-date servers might return erroneous results. A trace like that in Figure 7 would pinpoint the server responsible for the faulty data.

Faults could occur in the HTTP portion of the task as well. We describe the application of X-Trace to web traffic in the following section.

4.2 A web hosting site

Overview The second scenario that we consider is a web hosting service that allows users to post and share photographs. We deployed an open-source photo application in our network on an IBM Bladecenter. The front-end webserver host Apache and PHP. The photos, metadata, and comments are stored in a Postgres database. Also included are a cache and load-balancer. The photo site has attracted approximately 200 visitors a day for a period of two months.

For this site to support X-Trace, we implemented a reporting module for Apache, and one for Postgres. To support legacy web clients, we implemented an "X-Trace headers" module that inserted X-Trace headers into requests from the legacy clients.

X-Trace can be invoked by either end users or by the operator. End users can invoke X-Trace in two ways: by using an X-Trace-enabled web browser, or an X-Trace-equipped web page. We implemented an X-Trace toolbar for the Firefox web browser that puts X-Trace metadata in requests. We also implemented a Javascript/PHP library that added a feature to selected webpages in the site that let the user report problems via an HTML form. These reports were internally coupled with the X-Trace metadata of the user's request, enabling the network operator to match their complaint with a trace of their session. This is a powerful mechanism to detect semantic faults that would appear normal from the web site's perspective, such as stale pages or logic errors in a well formed response. This is not necessary for all faults, since many requests might generate anomalous task trees that can be analyzed with methods such as Pinpoint [9].

Tracing a request through the scenario The client application (i.e., *Firefox* with our X-Trace extension) creates a new X-Trace metadata and initializes its *TreeInfo* fields. It issues an annotated request to the front-end cache. This cache issues a report based on fields in the request and the X-Trace metadata. It calls `pushNext()` on the metadata, and forwards it on, possibly to other middleboxes such as load balancers that might also be on the path. When the Apache process on the front-end tier receives the request, it issues a report that includes the URL, status code, and time of the request.

The PHP-based photo software creates SQL statements to retrieve images and metadata from the back-end database. We modified this code to retrieve the X-Trace metadata from the array of HTTP headers and call `pushNext()` on it. The new metadata is propagated to the database by enclosing it in a SQL comment (i.e., `/* X-Trace:023A2E... */`). The query is sent to the database, which looks for embedded X-Trace metadata. It calls `xtr_report()` with the query as the payload of the report. When the webserver sends the response back to the client, it adds two headers to the response: one has the X-Trace metadata (in case it was generated by the webserver), and the other has a URL that the client can access to examine the trace of the request.

If any additional requests are generated because of that response (e.g., for images), the Firefox extension will use the same *TaskID*. For clients that don't support X-Trace, then each request (including images) will be considered independent.

Using X-Trace In this section we introduce several faults into the photo hosting site. These are based on first-hand experience that we had with our deployed system.

The first fault we consider is that of a malfunctioning PHP script on the front-end web servers. From the

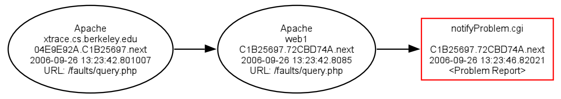


Figure 8: A request fault, annotated with user input

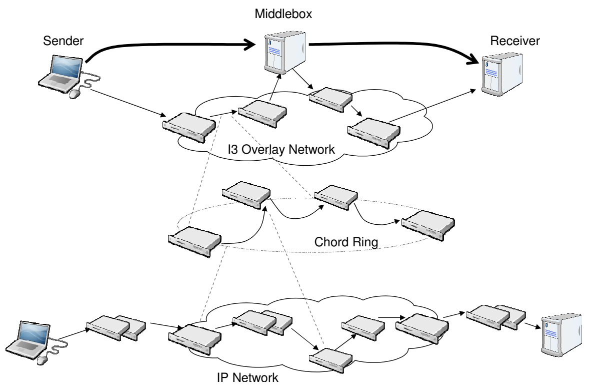


Figure 9: X-Trace on an I3 overlay scenario. A client and a server communicate over I3. Shown are the Chord network on top of which the I3 servers communicate, and the underlying IP network.

user's point of view, this could either be a fault in the PHP script, or a fault in the database. Examining Figure 8 shows immediately that the fault is the former–there are no reports from the database, pinpointing the problem to the PHP script. Figure 8 shows a square node that represents a problem report issued by the user, using the PHP/Javascript web problem reporting tool. In addition to triggering an alarm for the operator, the report node indicates which page caused the problem, in this case, `/faults/query.php`, located on `web1`.

Next, based on the Wikipedia example, we implemented a web cache that inadvertently returns stale images from its cache. Diagnosis in this case is simple. The request trace will include nodes up to and including the cache, but will not include the origin server.

The last fault we consider in this scenario is that of a malfunctioning web load balancer, which sends traffic to a server that doesn't contain the appropriate content. When users request pages from the site, they will sometimes get the pages they wanted, while other times they will get `404 File Not Found` errors. In both cases, the load balancer issues a report with the request URL. Successful requests also include reports from the working web server and backend database, while unsuccessful requests only include a report from the web server.

4.3 An overlay network

The third scenario we look at in some detail is an overlay network. Overlay networks are routing infrastructures that create communication paths by stitching together more than one end-to-end path on top of the underlying IP network. Overlays have been built to provide multicast [13], reliability [2], telephony [22], and data storage [25] services. It is difficult to understand the behavior and diagnose faults in these systems, as there are no tools or common frameworks to allow tracing of data connections through them.

In our example, we use the I3 overlay network [24]. For our purposes, it suffices to say that I3 provides a clean way to implement service composition, by interposing middleboxes on the communication path. The implementation of I3 we used runs on top of the Chord DHT [25], which provides efficient routing to flat identifiers and is an overlay network on its own.

We added X-Trace metadata to the I3 and Chord protocols, code to perform the `pushNext()` and `pushDown()` propagation operations, as well as calls to the X-Trace reporting library. The scenario topology is shown in Figure 9, and consists, at the highest layer, of a very simple protocol involving a sender, a receiver, and a middlebox interposed in the path by the sender. We used a toy protocol we called SNP – Simple Number Protocol – that is simply sending a number to the other party. The middlebox adds 10000 to any number it receives and forwards the request on, but it could also be, say, an HTTP proxy or a video transcoder. SNP also carries X-Trace metadata in its header. Each segment of the path in the SNP layer corresponds to a complete I3 path. Each I3 path, in turn, is formed by a combination of IP and Chord paths. Finally, each Chord path is formed by a combination of IP paths.

Tracing a message through the scenario In Figure 10(a) we show the reconstructed tree of operations given by X-Trace in a sample run of the scenario. This tree was generated from X-Trace reports by the visualization tool we developed. We deployed an I3 network consisting of 3 machines, each of which was also Chord node. The SNP client, receiver, and middlebox are on separate machines. We omit the IP report messages: all IP paths are one hop, since the machines were all on a switched LAN.

The SNP client sends a message to the the SNP receiver (see Figure 10), and it interposes the SNP middlebox on the path. The following is a detailed look at the transmission of a message in this scenario.

The SNP client creates a message, chooses a *TaskID* and includes X-Trace metadata in the SNP header. It chooses the I3 identifier stack $(ID_{middlebox}, ID_{server})$ as the destination (an identifier stack is simply a source-routed path in I3). The client calls `pushDown()`, copying the metadata into the I3 layer. Two more `pushDown()` operations copy it into the Chord and IP layers. The message is sent to the first I3 server, in this case at address 10.0.62.222. That server receives the message, and as it goes up the network stack, each layer generates and sends a report. The I3 server routes a message to the middlebox's I3 identifier, stored in the server 10.0.62.223. The I3 layer has a mapping between $ID_{middlebox}$ and the IP address 10.0.62.225. This message is delivered over IP to the I3 Client Library on that node, and then to the SNP Middlebox process.

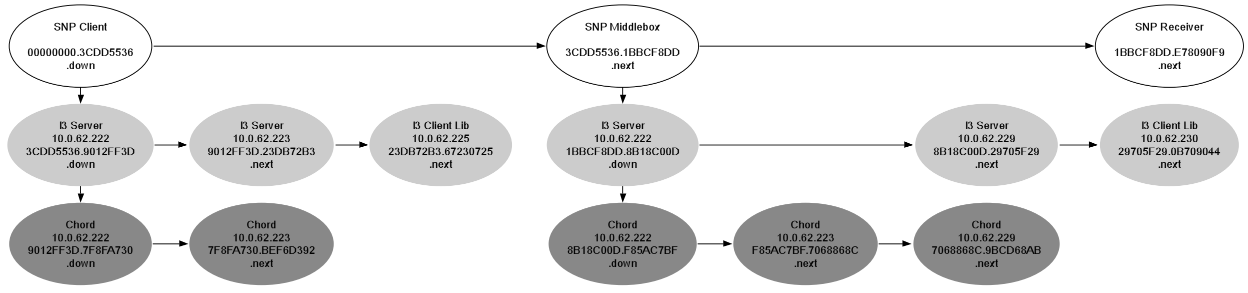
The middlebox receives the message and processes it, sending a report from each of its layers. It removes its I3 address from the identifier stack, leaving only the address of the server, ID_{server}. Like the client, it calls `pushNext()` on the identifier, and then `pushDown()` twice to propagate that ID to the Chord and IP layers. The next Chord node in the path, 10.0.62.223, receives the message and calls `pushNext()`. It sends a report, and then since there is no I3 layer, it simply forwards the message on. This process continues for the next I3 server, and finally the message is received by the receiver. At the receiver, we see a report from the I3 client library, and from the SNP application.

Using X-Trace In Figures 10(b), (c), and (d) we injected different types of faults and show how the resulting X-Trace tree detected them. We failed different components of the system that prevented the receiver from receiving the message. Normally it would be difficult or impossible for the sender to differentiate between these faults.

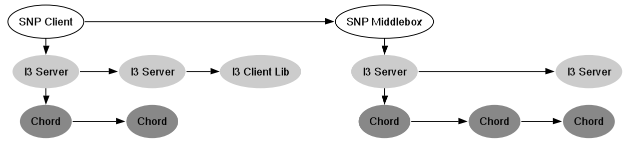
Fault 1: The receiver host fails In Figure 10(b) we simulated a crash in the receiver. I3 expires the pointer to the receiver machine after a timeout, and the result is that the message gets to the last I3 server before the receiver, but there is no report from either the SNP Receiver or I3 Client library at the receiver machine.

Fault 2: The middlebox process fails In Figure 10(c) we simulated a bug in the middlebox that made it crash upon receiving a specific payload and prevented it from forwarding the message. We see here that there is a report from the I3 Client library in the third I3 report node, but no report from the SNP middlebox or from any part of the tree after that. This indicates that the node was functioning at the time the message arrived. However, the lack of a report from the middlebox, coupled with no reports thereafter, points to the middlebox as the failure.

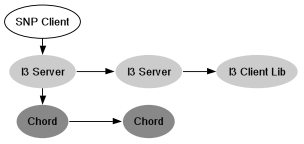
Fault 3: The middlebox host fails Finally, in Figure 10(d), we completely crashed the middlebox process. I3 expired the pointer to the machine, and we see the message stop at the last I3 server before the middlebox. The lack of any reports from the middlebox node, as well as no reports after the tree indicate that the entire node has failed.



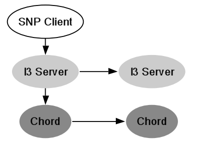
(a) Tree for normal operation



(b) Fault 1: The receiver host fails



(c) Fault 2: Middlebox process crash



(d) Fault 3: The middlebox host fails

Figure 10: (a)X-Trace tree corresponding to the i3 example scenario with a sender, a receiver, and a sender-imposed middlebox. (b), (c) and (d) correspond respectively to faults: a receiver crash, a middlebox process crash, and a crash of the entire middlebox machine.

5 Additional X-Trace Uses

Here we describe, in much briefer form, other scenarios where X-Trace could be used. This list isn't meant to be exhaustive, merely illustrative.

Tunnels: IPv6 and VPNs A tunnel is a network mechanism in which one data connection is sent in the payload of another connection. Two common uses are IPv6 and Virtual Private Networks (VPNs). Typically, it is not possible to trace a data path while it is in a tunnel. However, with X-Trace, the tunnel can be considered simply an additional layer. By calling `pushDown()`, the tunnel itself will contain the X-Trace identifier needed to send trace data about the tunnel to the sender.

ISP Connectivity Troubleshooting For consumers connecting to the Internet via an ISP, diagnosing connectivity problems can be quite challenging. ISP technical support staff members have to spend time trying to determine the location of faults that prevent the user from successfully connecting. Complicating this process is the myriad of protocols necessary to bring the user online: DHCP, PPPoE, DNS, firewalls, NATs, and higher layer applications such as E-mail and web caches.

By including X-Trace software in the client, as well as X-Trace support in the equipment at the premises, the ISP can determine the extent to which the user's traffic entered the ISP. This can help quickly identify the location of the problem, and thus reduce support costs.

Link layer tracing An enterprise network might want to trace the link layer, especially if there are highly lossy links such as a wireless access network. The effect of faults in these networks can have a profound effect on higher layer protocols, especially TCP [5]. Retrofitting X-Trace into Ethernet is not possible, due to its lack of extensibility. However, X-Trace metadata can easily be stored in a shim layer above Ethernet, but below other protocols. Since all of the hosts on a LAN make use of the same LAN protocol, it would be possible to deploy X-Trace enabled network devices within one enterprise without requiring higher level changes.

Development Tracing tasks is needed at one point or another in the development of distributed applications and protocols for debugging and verification. Like with standard logging subsystems, developers can integrate X-Trace into their applications. It is actually being used by the team developing DONA [16], a content-based routing scheme for the Internet.

6 Discussion

While X-Trace has many uses, it also has limitations. We discuss those here, as well as other interesting aspects.

Evaluation The examples and usage scenarios we implemented and described provide an indication of the usefulness of X-Trace in diagnosing and debugging distributed systems. However, the ultimate measure of success for X-Trace is when it can measurably help users and system administrators find problems faster than using ordinary tools, or problems that they wouldn't be able to find otherwise. We are working on moving in this direction, but such an analysis was beyond our means for this paper.

Report loss If the reporting infrastructure loses any reports, the effect to the graph will be the deletion of nodes and edges represented by that report. This might make it impossible to reconstruct the causal connections. In these cases, the reports sharing a common task identifier could be ordered temporally. Although not as descriptive, this linear graph might still pinpoint certain faults.

Managing report traffic The structure and complexity of an application's task trees have a strong bearing on the amount of report traffic generated by X-Trace nodes. We mention three mechanisms that can limit the volume of this traffic. *Sampling* can limit the number of requests that are tagged with X-Trace metadata to a rate specified by policy. A low sampling rate is ideal for "always-on" tracing used to get a picture of the behavior of the network. Differently from independent sampling at each node, using X-Trace, each "sample" is a complete task tree. Since X-Trace reports are delivered out-of-band, they can be *batched* and *compressed* before transmission. Within our network we have observed a 10x compression factor for X-Trace generated reports. Finally, *scoping* can be used to limit report generation to certain network layers, devices, or parts of the network. Layers such as IP generate many reports per request, since reports are generated on a per-packet basis. By limiting the scope of reports to those layers above IP, a smaller volume of reports is generated. Of course, if a fault is suspected at the network layer, the scope of reports could be widened to include IP packets of interest (say, from a client or subnet experiencing the observed problem). Currently, support for scoping is statically configured into the reporting infrastructure. Improving this support is considered future work.

Non-tree request structures The X-Trace metadata described in this work assumes that requests follow a tree structure. For the protocols and environments we considered, this assumption generally holds. However, other types of requests may not be captured. Examples are quorum protocols, or a controller which sends jobs to many working nodes and waits for all to complete. We are currently looking at extending the *TreeInfo* field to accommodate this.

Partial deployment Thus far, our discussion has focused on a comprehensive deployment of X-Trace throughout the network. However, even when X-Trace is partially deployed within one particular application or network layer, it still provides useful tracing benefits. For example, by integrating X-Trace into the I3 and Chord overlay networks, users of those system can track the mapping of I3 messages to Chord nodes. Alternatively, the developer of a middleware system could use X-Trace to follow requests from one node to another. In this spirit, researchers developing the DONA [16] project are making use of X-Trace to aid in the development of their new routing protocol.

Secondly, specific ADs can deploy X-Trace within their networks without requiring any cooperation or support from other ADs. For example, a service provider could deploy X-Trace at strategic points within their datacenter. This provides the service provider with the task tree within their network. We see the adoption of X-Trace following this partial deployment strategy.

Security Considerations It is important to discuss the potential for attacking the X-Trace infrastructure, as well as using that infrastructure to attack others.

First, one could mount an attack against an infrastructure that implements X-Trace by sending an inordinate amount of traffic with X-Trace metadata requesting reports. We argue that propagating metadata on its own is unlikely to become a bottleneck in this situation. Generating reports, however, could become a significant source of load. A simple defense is for each device to rate-limit the generation of reports. Still, malicious clients could get more than their fair share of the reporting bandwidth. If this becomes a problem, and filtering specific sources of reports becomes an issue, providers might start requiring capabilities in the options part of X-Trace metadata to issue reports.

Another possible attack with the reporting infrastructure is for a malicious user to send packets with X-Trace metadata, with the destination for reports set as another user. In the worst case, many network devices and hosts would send reports towards the attacked user. While this attack is possible, it will not have an exponential growth effect on the attacker's power, as legitimate reporting nodes will not place X-Trace metadata into X-Trace reports. Most important, however, is that we do not expect a large traffic of wide-area reports: as we describe in Section 2.3, we expect ADs to generate very few wire-area reports with pointers to detailed, independent stores for local reports within each AD. Lastly, this problem is more prevalent when the destination for reports are IP addresses. Using wire-area destinations like I3 or OpenDHT leverages these systems' denial of service prevention features. X-Trace keeps control of report generation rate and visibility with each report provider, which allows for defense mechanisms to be put in place.

7 Related Work

A number of tools focus on monitoring network status, aggregating data from many devices and layers. X-Trace differs from these tools in that it traces, across devices and layers, the actual paths taken by data messages, rather than trying to get snapshots of the network infrastructure as a whole. One such tool is `traceroute`, which traces IP network paths. SNMP [7] is a protocol that lets operators inspect instrumentation data from network devices such as packet counts and error conditions. HP Openview is an example of an enterprise-wide network management tool that makes use of SNMP data. Openview can coordinate views at different granularities, as well as coordinate network policy changes. Cisco Systems' Netflows [18] also provides device instrumentation, although at a finer granularity than SNMP.

Splunk [23] is a commercial solution that collects and indexes all logs of an IT installation, allowing administrators to interactively search these logs in a flexible way. With knowledge of common log formats it is usually possible to follow tasks through the logs with hints such as IP addresses, user names, and timestamps. However, this approach is unlikely to work across organizations, and is not guaranteed to have the relevant causal connections. Properly propagated X-Trace metadata added to logs can greatly enhance the search power of such a tool.

Hussain et al. [14] present a system for performing high-speed network traces at a large scale. The purpose of their work is to collect the data, process it according to anonymization policies, and make it available for multiple users. That work focuses on traffic in the network, and not on capturing causal connections between requests at different layers. Kompella et al. [15] present a service for collecting "cross-layer information". The focus of that work is on collecting control path state at different layers. Using the information their system collects, one could identify how failures at one layer impact other layers. X-Trace differs from that work in that we require widening the APIs at each layer, and focus on the datapath, rather than the control path.

The Application Response Measurement (ARM) [3] project annotates transactional protocols in corporate enterprises with identifiers. Devices in that system record start and end times for transactions, which can be reconciled offline. ARM targets the application layer, and its focus is to diagnose performance problems in nested transactions.

Pinpoint [9] detects faults in large, distributed systems. The authors modified J2EE middleware to capture the paths that component-based Java systems took through that middleware. They can mine collections of these paths to infer which components are responsible for causing faults. Our work focuses on recovering the task trees associated with multi-layer protocols, rather than the analysis of those recovered paths.

Aguilera et al., in [1], find anomalous behavior in distributed systems by treating each component as a black box, and inferring the operation paths by only looking at message traces. They present heuristics to recover the path given the timing relations among messages. A follow-up work, Pip [20] is an infrastructure for comparing actual and expected behavior of distributed systems by reasoning about paths through the application. They record paths by propagating path identifiers between components, and can specify recognizers for paths that deal with system communication structure, timing, resource consumption. Pip is targeted at a single distributed application, under the same AD, and does not capture cross-layer correlations. X-Trace is complementary to Pip in this sense. We believe that some of Pip's analysis can be performed on X-Trace's task trees.

Magpie [6] is a toolchain that works with events generated by operating system, middleware, and application instrumentation, correlates them, and produces representations of paths through a system by inferring causal relations from a total ordering of events. Instead of unique identifiers, Magpie relies on experts with deep knowledge about the system to construct a schema of how to correlate events in different components. Like X-Trace, they correlate lower level events with a higher level task, but focus mostly on a single system or on distributed systems that are highly instrumented in a compatible way.

The recent work in the AND and Constellation projects [4], defines the Leslie Graph as the graph representing the dependencies a distributed system's components. They use inference techniques to unobtrusively find correlations in traffic entering and leaving each node or service, and combine these findings in a network-wide graph. This graph is similar to, but different from our task trees: X-Trace produces deterministic traces of individual task executions, that are useful for examining their individual characteristics. In our ongoing work, we are looking into aggregating several task trees to determine aggregate behavior and dependencies.

Finally, Causeway [8] and SDI [19] provide mechanisms for automating metadata propagation within operating system and application structures, and could be used in some scenarios to ease X-Trace metadaa propagation.

8 Conclusions

Internet applications are becoming increasingly distributed and complex, taking advantage of new protocol layers and middlebox functionality. Current network diagnostic tools only focus on one particular protocol layer, and the insights they provide on the application cannot be shared between the user, service, and network operators. We propose X-Trace, a cross-layer, cross-application tracing framework designed to reconstruct the user's task tree. This framework enables X-Trace enabled nodes to encode causal connections necessary for rebuilding this tree. The trace data generated by X-Trace is published to a reporting infrastructure, ensuring that different parties can access it in a way that respects the visibility requirements of network and service operators.

We deployed and evaluated X-Trace in two concrete scenarios: a web hosting site and an overlay network. We found that with X-Trace, we were able to quickly identify the location of six injected faults. These faults were chosen because they are difficult to detect using current diagnostic tools.

The data generated by X-Trace instrumented systems can serve as the basis for more sophisticated analysis than the simple visualization and fault detection shown here. Using this data for new and existing algorithms [9, 20] is the object of our ongoing work. Given that the provider of reports ultimately controls how much data is generated, we are also investigating strategies to push filters on what to report as close to the sources of data as possible. For example, an AD could push a filter to all of its reporting daemons to not send reports on the IP layer.

Acknowledgments

We would like to thank our shepherd, Brad Karp, the anonymous NSDI reviewers, as well as Gautam Autekar, Matthew Caesar, Teemu Koponen, Armando Fox, Jeff Mogul, Sylvia Ratnasamy, Sean Rhea, Dick Sites, Nina Taft, and James Zhang for all the insightful comments and discussions.

This work was supported by NSF Grant CNS-0526016 ("NeTS-NBD: Collaborative Research: Intelligent and Adaptive Networking for the Next Generation Internet"), the California MICRO Program ("06-146: Enhanced Networking Dependability via Programmable Network Elements"), and with matching industrial support from Hewlett-Packard, Microsoft, and Sun Microsystems.

References

[1] AGUILERA, M. K., MOGUL, J. C., WIENER, J. L., REYNOLDS, P., AND MUTHITACHAROEN, A. Performance debugging for distributed systems of black boxes. In *Proc. SOSP '03* (New York, NY, USA, 2003), ACM Press.

[2] ANDERSEN, D., BALAKRISHNAN, H., KAASHOEK, F., AND MORRIS, R. Resilient overlay networks. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles* (New York, NY, USA, 2001), ACM Press.

[3] Application Response Measurement, http://www.opengroup.-org/tech/management/arm/.

[4] BAHL, P., BARHAM, P., BLACK, R., CHANDRA, R., GOLDSZMIDT, M., ISAACS, R., KANDULA, S., LI, L., MACCORMICK, J., MALTZ, D. A., MORTIER, R., WAWRZONIAK, M., AND ZHANG, M. Discovering dependencies for network management. In *Proc. V HotNets Workshop* (Nov. 2006).

[5] BALAKRISHNAN, H., PADMANABHAN, V. N., SESHAN, S., AND KATZ, R. H. A comparison of mechanisms for improving tcp performance over wireless links. In *Proc. SIGCOMM '96* (New York, NY, USA, 1996), ACM Press, pp. 256–269.

[6] BARHAM, P., DONNELLY, A., ISAACS, R., AND MORTIER, R. Using Magpie for Request Extraction and Workload Modeling. In *Proc. USENIX OSDI* (2004).

[7] CASE, J. D., FEDOR, M., SCHOFFSTALL, M. L., AND DAVIN, C. RFC 1157: Simple network management protocol (SNMP), May 1990.

[8] CHANDA, A., ELMELEEGY, K., COX, A. L., AND ZWAENEPOEL, W. Causeway: System support for controlling and analyzing the execution of multi-tier applications. In *Proc. Middleware 2005* (November 2005), pp. 42–59.

[9] CHEN, M., KICIMAN, E., FRATKIN, E., BREWER, E., AND FOX, A. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proc. International Conference on Dependable Systems and Networks* (2002).

[10] CROCKER, D. RFC 822: Standard for the format of ARPA Internet text messages, Aug. 1982.

[11] FIELDING, R., GETTYS, J., MOGUL, J., FRYSTYK, H., MASINTER, L., LEACH, P., AND BERNERS-LEE, T. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFC 2817.

[12] FONSECA, R., PORTER, G., KATZ, R. H., SHENKER, S., AND STOICA, I. IP options are not an option. Tech. Rep. UCB/EECS-2005-24, EECS Department, UC Berkeley, December 9 2005.

[13] HUA CHU, Y., RAO, S. G., SESHAN, S., AND ZHANG, H. A case for end system multicast. *IEEE Journal on Selected Areas in Communication (JSAC) 20*, 8 (2002).

[14] HUSSAIN, A., BARTLETT, G., PRYADKIN, Y., HEIDEMANN, J., PAPADOPOULOS, C., AND BANNISTER, J. Experiences with a continuous network tracing infrastructure. In *Proc. MineNet '05* (New York, NY, USA, 2005), ACM Press.

[15] KOMPELLA, R. R., GREENBERG, A., REXFORD, J., SNOEREN, A. C., AND YATES, J. Cross-layer visibility as a service. In *Proc. IV HotNets Workshop* (November 2005).

[16] KOPONEN, T., CHAWLA, M., CHUN, B.-G., ERMOLINSKIY, A., KIM, K. H., SHENKER, S., AND STOICA, I. A Data-Oriented (and Beyond) Network Architecture. In submission.

[17] MAZIÈRES, D. A toolkit for user-level file systems. In *USENIX Conference* (June 2001).

[18] Cisco NetFlow Services and Applications White Paper, http://www.cisco.com/go/netflow.

[19] REUMANN, J., AND SHIN, K. G. Stateful distributed interposition. *ACM Trans. Comput. Syst. 22*, 1 (2004), 1–48.

[20] REYNOLDS, P., KILLIAN, C., WIENER, J., MOGUL, J., SHAH, M., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In *Proc. USENIX NSDI* (May 2006).

[21] RHEA, S., GODFREY, B., KARP, B., KUBIATOWICZ, J., RATNASAMY, S., SHENKER, S., STOICA, I., AND YU, H. OpenDHT: a public DHT service and its uses. In *Proc. SIGCOMM '05* (New York, NY, USA, 2005), ACM Press, pp. 73–84.

[22] ROSENBERG, J., SCHULZRINNE, H., CAMARILLO, G., JOHNSTON, A., PETERSON, J., SPARKS, R., HANDLEY, M., AND SCHOOLER, E. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002.

[23] Splunk, http://www.splunk.com.

[24] STOICA, I., ADKINS, D., ZHUANG, S., SHENKER, S., AND SURANA, S. Internet indirection infrastructure. In *Proc. SIGCOMM '02* (New York, NY, USA, 2002), ACM Press, pp. 73–86.

[25] STOICA, I., MORRIS, R., KARGER, D., KAASHOEK, M. F., AND BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM '01* (New York, NY, USA, 2001), ACM Press, pp. 149–160.

[26] VIXIE, P. Extension Mechanisms for DSN (EDNS0). RFC 2671, Aug. 1999.

[27] Wikipedia Infrastructure, http://meta.wikimedia.org/wiki-/wikimedia_servers.

Friday: Global Comprehension for Distributed Replay

Dennis Geels[*], Gautam Altekar[‡], Petros Maniatis[ϕ], Timothy Roscoe[†], Ion Stoica[‡]
[*]*Google, Inc.,*[‡]*University of California at Berkeley,* [ϕ]*Intel Research Berkeley,* [†]*ETH Zürich*

Abstract

Debugging and profiling large-scale distributed applications is a daunting task. We present Friday, a system for debugging distributed applications that combines deterministic replay of components with the power of symbolic, low-level debugging and a simple language for expressing higher-level distributed conditions and actions. Friday allows the programmer to understand the collective state and dynamics of a distributed collection of coordinated application components.

To evaluate Friday, we consider several distributed problems, including routing consistency in overlay networks, and temporal state abnormalities caused by route flaps. We show via micro-benchmarks and larger-scale application measurement that Friday can be used interactively to debug large distributed applications under replay on common hardware.

1 Introduction

Distributed applications are complex, hard to design and implement, and harder to validate once deployed. The difficulty derives from the distribution of application state across many distinct execution environments, which can fail individually or in concert, span large geographic areas, be connected by brittle network channels, and operate at varying speeds and capabilities. Correct operation is frequently a function not only of single-component behavior, but also of the global collection of states of multiple components. For instance, in a message routing application, individual routing tables may appear correct while the system as a whole exhibits routing cycles, flaps, wormholes or other inconsistencies.

To face this difficulty, ideally a programmer would be able to debug *the whole application*, inspecting the state of any component at any point during a debugging execution, or even creating custom invariant checkers on global predicates that can be *globally* evaluated continuously as the system runs. In the routing application example, a programmer would be able to program her debugger to check continuously that no routing cycles exist across the running state of the entire distributed system, as easily as she can read the current state of program variables in typical symbolic debuggers.

Friday, the system we present in this paper, is a first step towards realizing this vision. Friday (1) captures the distributed execution of a system, (2) replays the captured execution trace within a symbolic debugger, and (3) extends the debugger's programmability for complex predicates that involve the *whole* state of the replayed system. To our knowledge, this is the first replay-based debugging system for unmodified distributed applications that can track arbitrary global invariants at the fine granularity of source symbols.

Capture and replay in Friday are performed using liblog [8], which can record distributed executions and then replay them consistently. Replay takes place under the control of a symbolic debugger, which provides access to internal application state. But simple replay does not supply the global system view required to diagnose emergent misbehavior of the application as a whole.

For global predicate monitoring or replayed applications (the subject of this paper), Friday combines the flexibility of symbolic debuggers on each replayed node, with the power of a general-purpose, embedded scripting language, bridging the two to allow a single global invariant checker script to monitor and control the global execution of multiple, distinct replayed components.

Contributions: Friday makes two contributions. First, it provides primitives for detecting events in the replayed system based on data (watchpoints) or control flow (breakpoints). These watchpoints and breakpoints are *distributed*, coordinating detection across all nodes in the replayed system, while presenting the abstraction of operating on the global state of the application.

Second, Friday enables users to attach arbitrary *commands* to distributed watchpoints and breakpoints. Friday gives these commands access to all application state as well as a persistent, shared store for saving debugging statistics, building behavioral models, or shadowing global state.

We have built an instance of `Friday` for the popular GDB debugger, using Python as the script language, though our techniques are equally applicable to other symbolic debuggers and interpreted scripting languages.

Applicability: Many distributed applications can benefit from `Friday`'s functionality, including both fully distributed systems (e.g., overlays, protocols for replicated state machines) and centrally managed distributed systems (e.g., load balancers, cluster managers, grid job schedulers). Developers can evaluate global conditions during replay to validate a particular execution for correctness, to catch inconsistencies between a central management component and the actual state of the distributed managed components, and to express and iterate behavioral regression tests. For example, with an IP routing protocol that drops an unusual number of packets, a developer might hypothesize that the cause is a routing cycle, and use `Friday` to verify cycle existence. If the hypothesis holds true, the developer can further use `Friday` to capture cycle dynamics (e.g., are they transient or long-lasting?), identify the likely events that cause them (e.g., router failures or congestion), and finally identify the root cause by performing step-by-step debugging and analysis on a few instances involving such events, all without recompiling or annotating the source code.

Structure: We start with background on `liblog` in Section 2. Section 3 presents the design and implementation of `Friday`, and also discusses the limitations of the system. We then present in Section 4 concrete usage examples in the context of two distributed applications: the Chord DHT [25], and a reliable communication toolkit for Byzantine network faults [26]. We evaluate `Friday` both in terms of its primitives and these case studies in Section 5. Finally, we present related work in Section 6 and conclude in Section 7.

2 Background: `liblog`

`Friday` leverages `liblog` [8] to deterministically and consistently replay the execution of a distributed application. We give a brief overview here.

`liblog` is a replay debugging tool for distributed `libc`- and POSIX C/C++-based applications on Linux/x86 computers. To achieve deterministic replay, each application thread records the side-effects of all nondeterministic system calls (e.g., `recvfrom()`, `select()`, etc.) to a local log. This is sufficient to replay the same execution, reproducing race conditions and non-deterministic failures, following the same code paths during replay, as well as the same file and network I/O, signals, and other IPC. `liblog` ensures causally consistent group replay, by maintaining Lamport clocks [16] during logging.

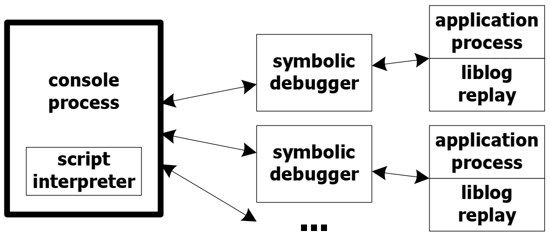


Figure 1: Overall architecture of `Friday`

`liblog` is incrementally deployable—it allows instrumented applications to communicate with applications that are not instrumented (e.g., DNS). `liblog` also supports replaying a subset of nodes without having to gather the logs of all nodes in the distributed system. Both incremental deployment and partial replay call for logging all incoming network traffic.

Finally, `liblog`'s library-based implementation requires neither virtualization nor kernel additions, resulting in a small per-process CPU and memory footprint. It is lightweight enough to comfortably replay 60 nodes on a Pentium D 2.8GHz machine with 2GB of RAM. We have also built a proof-of-concept cluster-replay mechanism that can scale this number with the size of the replay cluster to thousands of nodes.

While `liblog` provides the programmer with the basic information and tools for debugging distributed applications, the process of tracking down the root cause of a particular problem remains a daunting task. The information presented by `liblog` can overwhelm the programmer, who is put, more often than not, in the position of finding a "needle in the haystack." `Friday` enables the programmer to prune the problem search space by expressing complex global conditions on the state of the whole distributed application.

3 Design

`Friday` presents to users a central debugging console, which is connected to replay processes, each of which runs an instance of a traditional symbolic debugger such as GDB (see Figure 1). The console includes an embedded script language interpreter, which interprets actions and can maintain central state for the debugging session. Most user input is passed directly to the underlying debugger, allowing full access to data analysis and control functions. `Friday` extends the debugger's commands to handle distributed breakpoints and watchpoints, and to inspect the whole system of debugged processes.

3.1 Distributed Watchpoints and Breakpoints

Traditional watchpoints allow a symbolic debugger to react—stop execution, display values, or evaluate a predicate on the running state—when the process updates a particular variable location, named via a memory address or a symbolic name from the application's source.

Friday's distributed watchpoints extend this functionality for variables and expressions from multiple nodes in the replayed distributed application. For example, a programmer debugging a ring network can use Friday to watch a variable called successor on all machines by specifying "watch successor" or on a single machine (here, #4) with "4 watch successor". The command "[<node number>, ...] watch <variable> [...]" specifies both a set of nodes on which to watch variables (all by default), and a set of variables to watch. The node numbering is private to Friday; to identify a particular node by another identifier such as an IP address, an appropriate mapping can be provided (Section 3.2).

Distributed breakpoints in Friday have a similar flavor. Like traditional breakpoints, they allow the debugger to react when the debugged process executes a particular instruction, specified as a source line number or a function name. Friday allows the installation of such breakpoints on one, several, or all replayed nodes.

3.1.1 Implementation

Friday implements distributed watchpoints and breakpoints by setting local instances on each replay process and mapping their individual numbers and addresses to a global identifier. These maps are used to rewrite and forward disable/enable requests to a local instance, and also to map local events back to the global index when executing attached commands.

Local breakpoints simply use GDB breakpoints, which internally either use debugging registers on the processor or inject trap instructions into the code text. In contrast, Friday implements its own mechanism for local watchpoints. Friday uses the familiar technique of write-protecting the memory page where the value corresponding to a given symbol is stored [29]. When a write to the variable's page occurs, the ensuing SEGV signal is intercepted, leading Friday to unprotect the page and completes the write, before evaluating any state manipulation scripts attached to the watchpoint.

This implementation can give rise to *false positives* when an unwatched variable sharing the page with a watchpoint is written. The more densely populated a memory page, the more such false positives occur. We decided that protection-based watchpoints are preferable to alternative implementations.

We explored but rejected four alternatives: hardware watchpoints, single stepping, implementation via breakpoints, and time-based sampling.

Hardware watchpoints are offered by many processor architectures. They are extremely efficient, causing essentially no runtime overhead, but most processors have small, hard limits on the number of watchpoint registers (a typical value is 8), as well as on the width of the watched variable (typically, a single machine word). For instance, watching for changes to a variable across tens of replayed nodes would not be possible if the replay machine has only 8 watchpoint registers. These limits are too restrictive for distributed predicates; however, we have planned a hybrid system that uses hardware watchpoints as a cache for our more flexible mechanism.

Single-stepping, or *software watchpoints*, execute one machine instruction and check variable modifications at each step. Unfortunately, single-stepping is prohibitively slow—we compare it to our method in Section 5.4 and demonstrate that it is a few thousand times slower.

Local breakpoints can emulate watchpoints by identifying the points where the watched variable could be modified and only checking for changes there. When this identification step is accurate the technique is highly efficient, but unfortunately it requires comprehensive knowledge of the program code and is prone to mistakes.

Periodic sampling of watched variables (e.g., every k logical time ticks) enables a trade-off between replay speedup and watchpoint accuracy: it is potentially faster than all the techniques described above, but it may be difficult to pinpoint value changes. Combined with replay checkpointing and backtracking, it might prove a valuable but not complete alternative.

3.1.2 Implementation Complexity

Building a new watchpoint mechanism in Friday required reconstructing some functionality normally provided by the underlying symbolic debugger, GDB. Namely, debuggers maintain state for each watched expression, including the stack frame where the variable is located (for local variables) and any mutable subexpressions whose modification might affect the expression's value. For example, a watchpoint on srv->successor->addr should trigger if the pointers srv or srv->successor change, pointing the expression to a new value. Because GDB does not expose this functionality cleanly, we replicated it in Friday.

Also, the new watchpoint mechanism conflicts with GDB's stack maintenance algorithms. Friday's manipulation of memory page protection on the stack (Section 3.1.1) conflicts with GDB's initialization tasks when calling application functions, causing mprotect failures. To resolve the conflict, we replace GDB's calling facilities with our own, manipulating the application's PC directly, thereby complicating GDB's breakpoint maintenance. Thankfully, these complications are not triggered by any of our case studies presented in this paper.

3.2 Commands

The second crucial feature of Friday is the ability to view and manipulate the distributed state of replayed nodes. These actions can either be performed interactively or triggered automatically by watchpoints or breakpoints. Interactive commands such as `backtrace` and `set` are simply passed directly to the named set of debugger processes. They are useful for exploring the distributed state of a paused system.

In contrast, automated commands are written in a scripting language for greater expressiveness. These commands are typically used to maintain additional views of the running system to facilitate statistics gathering or to reveal complex distributed (mis)behaviors.

Friday commands can maintain their own arbitrary debugging state, in order to gather statistics or build models of global application state. In the examples below, `emptySuccessors` and `nodes` are debugging state, declared in Friday via the `python` statement; e.g., `python emptySuccessors = 0`. This state is shared among commands and is persistent across command executions.

Friday commands can also read and write variables in the state of any replayed process, referring to symbolic names exposed by the local GDB instances. To simplify this access, Friday embeds into the scripting language appropriate syntax for calling functions and referencing variables from replayed processes. For example, the statement "`@4(srv.successor) == @6(srv.predecessor)`" compares the successor variable on node 4 to the predecessor variable on node 6. By omitting the node specifier, the programmer refers to the state on the node where a particular watchpoint or breakpoint was triggered. For example, the following command associated with a watchpoint on `srv.successor` increments the debugging variable `emptySuccessors` whenever a successor pointer is set to `null`, and continues execution:

```
if not @(srv.successor):
   emptySuccessors++
cont
```

For convenience, the node where a watchpoint or breakpoint was triggered is also accessible within command scripts via the `__NODE__` metavariable, and all nodes are available in the list `__ALL__`. For example, the following command, triggered when a node updates its application-specific identifier variable `srv.node.id`, maintains the global associative array `nodes`:

```
nodes[@(srv.node.id)] = __NODE__
cont
```

Furthermore, Friday provides commands with access to the logical time kept by the Lamport clock exported by liblog, as well as the "real" time recorded at each log event. Because liblog builds a logical clock that is closely correlated with wall clock during trace acquisition, these two clocks are usually closely synchronized. Friday exposes the global logical clock as the `__LOGICALCLOCK__` metavariable and node i's real clock at the time of trace capture as `@i(__REALCLOCK__)`.

Similarly to GDB commands, our language allows setting and resetting distributed watchpoints and breakpoints from within a command script. Such *nested* watchpoints and breakpoints can be invaluable in selectively picking features of the execution to monitor in reaction to current state, for instance to watch a variable only in between two breakpoints in an execution. This can significantly reduce the impact of false positives, by enabling watchpoints only when they are relevant.

3.2.1 Language Choice

The Friday commands triggered by watchpoints and breakpoints are written in Python, with extensions for interacting with distributed application state.

Evaluating Python inside Friday is straightforward, because the console is itself a Python application, and dynamic evaluation is well supported. We chose to develop Friday in Python for its high-level language features and ease of prototyping; these benefits also apply when writing watchpoint command scripts.

We could have used the application's native language (C/C++), in much the same way that C is used in IntroVirt [11]. Such an approach would allow the programmer to inline predicate code in the language of the application, thereby simplifying the interface between C/C++ constructs and a higher-level language. It would also eliminate the need to rely on GDB and Python for breakpoint/watchpoint detection and predicate evaluation, thereby reducing IPC-related overhead during replay. Unfortunately, this option calls for duplicating much of the introspection functionality (e.g., inspection of stack variables) already offered by GDB, and requires recompiling/reloading a C/C++ predicate library each time the user changes a predicate; we wanted to support a more interactive usage model.

At the opposite end of the spectrum, we could have used GDB's "command list" functionality to express distributed watchpoints and breakpoints. Unfortunately GDB commands lack the expressiveness of Python, such as its ability to construct new data structures, as well as the wealth of useful libraries. Using a general-purpose scripting framework like Python running at the console afforded us much more flexibility.

3.2.2 Syntax

When a distributed command is entered, Friday examines every statement to identify references to the target application state. These references are specified with the syntax `@<node>(<symbol>[=<value>])` where the `node` defaults to that which triggered the breakpoint or watchpoint. These references are replaced with calls to internal functions that read from or write to the application using GDB commands `print` and `set`, respectively. Metavariables such as `__LOGICALCLOCK__` are interpolated similarly. Furthermore, `Friday` allows commands to refer to application objects on the heap whose symbolic names are not within scope, especially when stopped by a watchpoint outside the scope within which the watchpoint was defined. Such pointers to heap objects that are not always nameable can be passed to watchpoint handlers as parameters at the time of watchpoint definition, much like continuations (see Section 4.2.1 for a detailed example). The resulting statements are compiled, saved, and later executed within the global `Friday` namespace and persistent command local namespace.

If the value specified in an embedded assignment includes keyed printf placeholders, i.e., `%(<name>)<fmt>`, the value of the named Python variable will be interpolated at assignment time. For example, the command

```
tempX = @(x)
tempY = @other(y)
@(x=%(tempY)d)
@other(y=%(tempX)d)
```

swaps the values of integer variables `x` at the current node and `y` at the node whose number is held in the python variable `other`.

Commands may call application functions using similar syntax:

```
@<node>(<function>(<arg>,....))
```

These functions would fail if they attempted to write to a memory page protected by `Friday`'s watchpoint mechanism, so `Friday` conservatively disables all watchpoints for that replay process during the function call. Unfortunately that precaution may be very costly (see Section 5). If the user is confident that a function will not modify protected memory, she may start the command with the `safe` keyword, which instructs `Friday` to leave watchpoints enabled. This option is helpful, for example, if the invoked function only modifies the stack, and watchpoints are only set on global variables.

The value returned by GDB using the `@()` operator must be converted to a Python value for use by the command script. `Friday` understands strings (type `char*` or `char[]`), and coerces pointers and all integer types to Python `long` integers. Any other type, including any structs and class instances, are extracted as a tuple containing their raw bytes. This solution allows simple identity comparisons, which was sufficient for all useful case studies we have explored so far.

Finally, our extensions had to resolve some keyword conflicts between GDB and Python, such as `cont` and `break`. For example, within commands `continue` refers to the Python keyword whereas `cont` to GDB's keyword. In the general case, we can prefix the keyword `gdb` in front of GDB keywords within commands.

3.3 Limitations

We have used `Friday` to debug large distributed applications. Though still a research prototype with rough edges and only a rudimentary user interface, we have found `Friday` to be a powerful and useful tool; however, it has several limitations that potential users should consider.

We start with limitations that are inherent to `Friday`. First, false positives can slow down application replay. False positive rates depend on application structure and dynamic behavior, which vary widely. In particular, watching variables on the stack can slow `Friday` down significantly. In practice we have circumvented this limitation by recompiling the application with directives that spread the stack across many independent pages of memory. Though this runs at odds with our goal of avoiding recompilation, it is only required once per application, as opposed to requiring recompilations every time a monitored predicate or metric must change. Section 5 has more details on `Friday` performance.

The second `Friday`-specific limitation involves replaying from the middle of a replay trace. Some `Friday` predicates build up their debugging state by observing the dynamic execution of a replayed application, and when starting from a checkpoint these predicates must rebuild that state through observation of a static snapshot of the application at that checkpoint. This process is straightforward for the applications we study in Section 4, but it may be more involved for applications with more complex data structures. We are working on a method for adding debugging state to `liblog` checkpoints at debug time, to avoid this complexity.

Thirdly, although we have found that `Friday`'s centralized and type-safe programming model makes predicates considerably simpler than the distributed algorithms they verify, `Friday` predicates often require some debugging themselves. For example, Python's dynamic type system allows us to refer to application variables that are not in dynamic scope, causing runtime errors.

Beyond `Friday`'s inherent limitations, the system inherits certain limitations from the components on which it depends. First, an application may copy a watched variable and modify the copy instead of the original, which GDB is unable to track. This pattern is common, for example, in the collection templates of the C++ Standard Template Library, and requires the user of GDB (and consequently `Friday`) to understand the program well enough to place watchpoints on all such copies. The problem is exacerbated by the difficulty of accessing these copies, mostly due to GDB's inability to place breakpoints on STL's many inlined accessor functions.

A second inherited limitation is unique to stack-based variables. As with most common debuggers, we have no solution for watching stack variables in functions that have not yet been invoked. To illustrate, it is difficult to set up ahead of time a watchpoint on the command line argument variable `argv` of the `main` function across all nodes before we have entered the `main` at all nodes. Nested watchpoints are a useful tool in that regard.

Finally, `Friday` inherits `liblog`'s large storage requirements for logs and an inability to log or replay threads in parallel on multi-processor machines.

4 Case Studies

In this section, we present use cases for the new distributed debugging primitives presented above. First, we look into the problem of consistent routing in the i3/Chord DHT [24], which has occupied networking and distributed research literature extensively. Then we turn to debugging `Tk`, a reliable communication toolkit [26], and demonstrate sanity checking of disjoint path computation over the distributed topology, an integral part of many secure-routing protocols. For brevity, most examples shown omit error handling, which typically adds a few more lines of Python script.

4.1 Routing Consistency

In this section, we describe a usage scenario in which `Friday` helps a programmer to drill down on a reported bug with i3/Chord. The symptom is the loss of a value stored within a distributed hash table: a user who did a `put(key,value)`, doing a `get(key)` later did not receive the `value` she put into the system before. We describe a debugging session for this scenario and outline specific uses of `Friday`'s facilities.

Our programmer, Frieda, starts with a set of logs given to her, and with the knowledge that two requests, a `put` and a `get` that should be consistent with each other appear to be inconsistent: the `get` returns something other than what the `put` placed into the system.

4.1.1 Identifying a distributed bug

Frieda would probably eliminate non-distributed kinds of bugs first, by establishing for instance that a node-local store does not leak data. To do that, she can monitor that the two requests are handled by the same node, and that the node did not lose the key-value pair between the two requests.

```
py getNode = None
py putNode = None
break process_data
command
  if @(packet_id) == failing_id:
    if is_get(@(packet_header)):
      getNode = @(srv.node.id)
    else:
      putNode = @(srv.node.id)
end
```

This breakpoint triggers every time a request is forwarded towards its final destination. Frieda will interactively store the appropriate message identifier in the Python variable `failing_id` and define the Python method `is_get`. At the end of this replay session, the variables `getNode` and `putNode` have the identifiers of the nodes that last serviced the two requests, and Frieda can read them through the `Friday` command line. If they are the same, then she would proceed to debug the sequence of operations executed at the common node between the `put` and the `get`. However, for the purposes of our scenario we assume that Frieda was surprised to find that the `put` and the `get` were serviced by different nodes. This leads her to believe that the system experienced *routing inconsistency*, a common problem in distributed lookup services where the same lookup posed by different clients at the same time receives different responses.

4.1.2 Validating a Hypothesis

The natural next step for Frieda to take is to build a map of the consistent hashing offered by the system: which part of the identifier space does each node think it is responsible for? If the same parts of the identifier space are claimed by different nodes, that might explain why the same key was serviced by different nodes for the `put` and the `get` requests. Typically, a node believes that its immediate successor owns the range of the identifier space between its own identifier and that of its successor.

The following breakpoint is set at the point that a node sets its identifier (which does not change subsequently). It uses the `ids` associative array to map `Friday` nodes to Chord IDs.

```
py ids = {}
break chord.c:58
command
  ids[__NODE__] = @((char*)id)
  cont
end
```

Now Frieda can use this information to check the correct delivery of requests for a given key as follows:

```
break process.c:69
command
  if @(packet_id) != failing_id:
    cont
  for peer in __ALL__:
    @((chordID)_liblog_workspace =
        atoid("%(ids[peer])s"))
    if @(is_between((chordID*)&_liblog_workspace,
          @(packet_id), &successor->id)):
      print "Request %s misdelivered to %s" %
                  (@(packet_id), @(successor->id))
      break
  cont
end
```

This breakpoint triggers whenever a node with ID `srv.node.id` believes it is delivering a packet with destination ID `packet_id` to its rightful destination: the node's

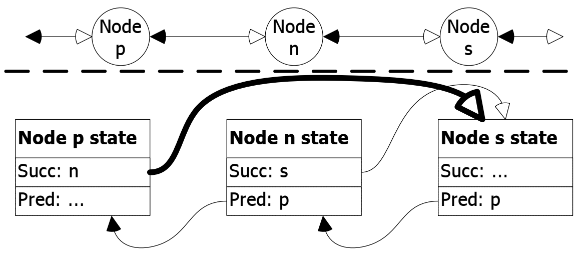


Figure 2: At the top, we show what node n believes the ring topology to be around it. At the bottom, we see the relevant state as stored by the involved nodes n, s and p. The thick routing entry from node p to s is inconsistent with n's view of the ring, indicating a source of routing consistency problems.

immediate successor with ID `srv.successor->id`, such that `packet_id` is in between identifiers `srv.node.id` and `srv.successor->id`. When that happens, this command figures out if the request in question is one of Frieda's problem requests, and if so, it finds out if there is a node that should be receiving the packet instead.

This check uses the native Chord function `atoid` to load the peer's ID into application scratch space (`_liblog_workspace`) and then invokes the Chord function `is_between` to perform the range check. Both of these functionalities could have been duplicated instead in Python, if Frieda suspected their implementation as the source of the problem.

This breakpoint command is a simple instance of a very powerful construct: a global index of all nodes in the system is very easy for Friday to construct at replay time but difficult or impossible to collect reliably and efficiently at runtime. Doing so would require transmitting each update (node insertion, etc.) to all nodes, presumably while all other communication is disabled to avoid inconsistencies. These updates would be expensive for large networks and could fail due to transient network conditions. Conversely, Friday can maintain a global view of the whole logged population, even if the nodes themselves could not talk to each other at runtime.

4.1.3 Searching for a Root Cause

The identified inconsistency told Frieda that she has a problem. Most likely, it tells her that part of her purported ring topology looks like Figure 2, in which the culprit node, p, believes its successor to be node s and delivers anything between identifiers p and s to s for processing, where instead all requests for identifiers between p and n belong to node n instead.

To dig deeper, Frieda can monitor ring consistency more closely, for instance by ensuring that ring edges are symmetric. Checking that successor/predecessor consistency conditions hold at all times is unnecessary. Instead, it is enough to check the conditions when a successor or predecessor pointer changes, and only check those specific conditions in which the changed pointers participate. Frieda can encode this in Friday as follows:

```
watch srv.successor
command
  successor_id = @(srv.successor->id)
  if @(srv.node.id) !=
      @nodes[successor_id](srv.predecessor->id):
    print __NODE__, "'s successor link is asymmetric."
end
```

and symmetrically for the predecessor's successor. This would catch, for instance, the problem illustrated in Figure 2, which caused Frieda's problem.

4.1.4 How Often Is The Bad Thing Happening?

Such inconsistencies occur transiently even when the system operates perfectly while an update occurs, e.g., when a new node is inserted into the ring. Without transactional semantics across all involved nodes in which checks are performed only before or after a transition, such warnings are unavoidable. Frieda must figure out whether this inconsistency she uncovered occurs most of the time or infrequently; that knowledge can help her decide whether this is a behavior she should mask in her Chord implementation (e.g., by sending redundant `put` and `get` requests) or fix (e.g., by ensuring that nodes have agreed on their topological neighborhood before acting on topology changes).

In Friday, Frieda can compute the fraction of time during which the ring topology lies in an inconsistent state. Specifically, by augmenting the monitoring statements from Section 4.1.3, she can instrument transitions from consistent to inconsistent state and back, to keep track of the time when those transitions occur, and averaging over the whole system.

```
watch srv.successor, srv.predecessor
command
  myID = @(srv.node.id)
  successorID = @(srv.successor->id)
  predecessorID = @(srv.predecessor->id)
  if not (@nodes[successorID](srv.predecessor->id)
     == @nodes[predecessorID](srv.successor->id)
     == myID): #inconsistent?
    if consistent[myID]:
      consistentTimes +=
          (@(__REALCLOCK__) - lastEventTime[myID])
      consistent[myID] = False
      lastEventTime[myID] = @(__REALCLOCK__)
  else: # converse: consistent now
    if not consistent[myID]:
      inconsistentTimes +=
          @((__REALCLOCK__) - lastEventTime[myID])
      consistent[myID] = True
      lastEventTime[myID] = @(__REALCLOCK__)
  cont
end

py consistent = {}
py lastEventTime = {}
py consistentTimes = inconsistentTimes = 0
```

This example illustrates how to keep track of how much time each replayed machine is in the consistent or inconsistent state, with regards to its ring links. The monitoring specification keeps track of the amounts of time node i is consistent or inconsistent in the debugging counters `consistentTimes` and `inconsistentTimes`, respectively. Also, it remembers when the last time a node switched to consistency or inconsistency in the debugging hash tables `consistent` and `inconsistent`, respectively. When the distributed commands are triggered, if the node is now inconsistent but was not before (the last time of turning consistent is non-empty), the length of the just-ended period of consistency is computed and added to the thus-far sum of consistency periods. The case for inconsistency periods is symmetric and computed in the "else" clause.

Periodically, or eventually, the relevant ratios can be computed as the ratio of inconsistent interval sums over the total time spent in the experiment, and the whole system might be characterized taking an average or median of those ratios.

4.1.5 State Oscillation

If Frieda finds that most of the time such inconsistencies exist, she may decide this is indeed a bug and move to fix it by ensuring a node blocks requests while it agrees on link symmetry with its immediate neighbors.

In the unhappy case in which Frieda's system is indeed intended to have no such inconsistencies (i.e., she has already written the code that causes nodes to agree on link symmetry), she would like to determine what went wrong. She can do by testing a series of hypotheses.

One such hypothesis—which is frequently the case of inconsistencies in a broad range of distributed applications—is a network link that, whether due to high loss rates or intermittent hardware failure, makes a machine repeatedly disappear and reappear to its neighbor across the link. This oscillation may cause routes through the nodes to flap to backup links, or even create routing wormholes and black holes. Frieda can analyze the degree of oscillation in her network with the following simple `Friday` breakpoint commands.

```
break remove_finger
command
  finger = @(f->node.addr)  # f is formal parameter
  events = routeEvents[@(srv.node.addr)]
  if finger not in events:
    events[finger] = []
  events[finger].append(("DOWN",__LOGICALCLOCK__))
  cont
end

break insert_finger
command
  finger = @(addr) # addr is formal parameter
  events = routeEvents[@(srv.node.addr)]
  if finger in events:
    lastEvent,time = events[finger][-1]
    if lastEvent == "DOWN":
      events[finger].append(("UP",__LOGICALCLOCK__))
  cont
end
```

The first command adds a log entry to the debugging table `routeEvents` (initialized elsewhere) each time a routing peer, or *finger*, is discarded from the routing table. The second command adds a complementary log entry if the node is reinserted. The two commands are asymmetric because `insert_finger` may be called redundantly for existing fingers, and also because we wish to ignore the initial insertion for each finger. The use of virtual clocks here allows us to correlate log entries across neighbors.

4.2 A Reliable Communication Toolkit

In the second scenario, we investigate `Tk` [26], a toolkit that allows nodes in a distributed system to communicate reliably in the presence of k adversaries. The only requirement for reliability is the existence of at least k disjoint paths between communicating nodes. To ensure this requirement is met, each node pieces together a global graph of the distributed system based on path-vector messages and then computes the number of disjoint paths from itself to every other node using the max-flow algorithm. A bug in the disjoint path computation or path-vector propagation that mistakenly registers k or more disjoint paths would seriously undermine the security of the protocol. Here we show how to detect such a bug.

4.2.1 Maintaining a Connectivity Graph

When performing any global computation, including disjoint-path computation, a graph of the distributed system is a prerequisite. The predicate below constructs such a graph by keeping track of the connection status of each node's neighbors.

```
py graph = zero_matrix(10, 10)

break server.cpp:355
command
  neighbor_pointer = "(*(i->_M_node))"
  neighbor_status_addr =
    @(&(%(neighbor_pointer)s->status))

  # Set a watchpoint dynamically
  watchpoint(["*%d" % neighbor_status_addr],
    np=@(%(neighbor_pointer)s))
  command
    status = @((((Neighbor*)(%(np)d))->status))
    neighbor_id = @((((Neighbor*)(%(np)d))->id))
    my_id = @(server->id)
    if status > 0:
      graph[my_id][neighbor_id] = 1
      compute_disjoint_paths() # Explained below.
    cont
  end
  cont
end
```

This example showcases the use of nested watchpoints, which are necessary when a watchpoint must be set at a specific program location. In this application, a neighbor's connection status variable is available only when the neighbor's object is in scope. Thus, we place a breakpoint at a location where all neighbor objects are enumerated, and as they are enumerated, we place a watchpoint on each neighbor object's connection status variable. When a watchpoint fires, we set the corresponding flag in an adjacency matrix.

A connection status watchpoint can be triggered from many programs locations, making it hard to determine what variables will be in scope for use within the watchpoint handler. In our example, we bind a watchpoint handler's `np` argument to the corresponding neighbor object pointer, thereby allowing the handler to access the neighbor object's state even though a pointer to it may not be in the application's dynamic scope.

4.2.2 Computing Disjoint Paths

The following example checks the toolkit's disjoint path computation by running a centralized version of the disjoint path algorithm on the global graph created in the previous example. The predicate records the time at which the k-path requirement was met, if ever. This timing information can then be used to detect disagreement between `Friday` and the application or to determine node convergence time, among other things.

```
py time_Friday_found_k_paths = zero_matrix(10, 10)

def compute_disjoint_paths():
  my_id = @(server->id)
  k = @(server->k)
  for sink in range(len(graph)):
    Friday_num_disjoint_paths =
      len(vertex_disjoint_paths(graph, my_id, sink))
    if Friday_num_disjoint_paths >= k:
      time_Friday_found_k_paths[my_id][sink] =
        __VCLOCK__
```

The disjoint path algorithm we implemented in `vertex_disjoint_paths`, not shown here, employs a brute force approach—it examines all k combinations of paths between source and destination nodes. A more efficient approach calls for using the max-flow algorithm, but that's precisely the kind of implementation complexity we wish to avoid. Since predicates are run offline, `Friday` affords us the luxury of using an easy-to-implement, albeit slow, algorithm.

4.3 Discussion

As the preceding examples illustrate, the concept of an invariant may be hard to define in a distributed system. So-called invariants are violated even under correct operation for short periods of time and/or within subsets of the system. `Friday`'s embedded interpreter allows programmers to encode what it means for a particular system to be "too inconsistent".

By gaining experience with the patterns frequently used by programmers to track global system properties that are transiently violated, we intend to explore improved high-level constructs for expressing such patterns as part of our future work.

The Python code that programmers write in the process of debugging programs with `Friday` can resemble the extra, temporary code added inline to systems when debugging with conventional tools: in cases where simple assertions or logging statements will not suffice, it is common for programmers to insert complex system checks which then trigger debugging code to investigate further the source of the unexpected condition.

In this respect, `Friday` might be seen as a system for "aspect-oriented debugging", since it maintains a strict separation between production code and diagnostic functionality. The use of a scripting language rather than C or C++ makes writing debugging code easier, and this can be done after compilation of the program binaries. However, `Friday` also offers facilities not feasible with an on-line aspect-oriented approach, such as access to global system state.

It has often been argued that debugging code should never really be disabled is a production distributed system. While we agree with this general sentiment, in `Friday` we draw a more nuanced distinction between code which is best executed all the time (such as configurable logging and assertion checks), and that which is only feasible or useful in the context of offline debugging. The latter includes the global state checks provided by `Friday`, something which, if implemented inline, would require additional inter-node communication and library support.

5 Performance

In this section, we evaluate the performance of `Friday`, by reporting its overhead on fundamental operations (micro-benchmarks) and its impact on the replay of large distributed applications. Specifically, we evaluate the effects of false positives, of debugging computations, and of state manipulations in isolation, and then within replays of a routing overlay.

For our experiments we gathered logs from a 62-node i3/Chord overlay running on PlanetLab [3]. After the overlay had reached steady state, we manually restarted several nodes each minute for ten minutes, in order to force interesting events for the Chord maintenance routines. No additional lookup traffic was applied to the overlay. All measurements were taken from a 6 minute stretch in the middle of this turbulent period. The logs were replayed in `Friday` on a single workstation with a Pentium D 2.8GHz dual-core x86 processor and 2GB

Benchmark	Latency (ms)
False Positive	13.2
Null Command	15.6
Value Read	15.9
Value Write	15.9
Function Call	26.1
Safe Call	16.5

Table 1: Micro-benchmarks - single watchpoint

RAM, running the Fedora Core 4 OS with version 2.6.16 of the Linux kernel.

5.1 Micro-benchmarks

Here we evaluate `Friday` on six micro-benchmarks that illustrate the overhead required to watch variables and execute code on replayed process state. Table 1 contains latency measurements for the following operations:

- *False Positive*: A watchpoint is triggered by the modification of an unwatched variable that occupies the same memory page as the watched variable.
- *Null Command*: The simplest command we can execute once a watchpoint has passed control to `Friday`. The overhead includes reading the new value (8 bytes) of the watched variable and evaluating a simple compiled Python object.
- *Value Read*: A single fetch of a variable from one of the replayed processes. The overhead involves contacting the appropriate GDB process and reading the variable's contents.
- *Value Write*: Updates a single variable in a single replayed process.
- *Function Call*: The command calls an application function that returns immediately. All watchpoints (only one in this experiment) must be disabled before, and re-enabled after the function call.
- *Safe Call*: The command is marked "safe" to obviate the extra watchpoint management.

These measurements indicate that the latency of handling the segmentation faults dominates the cost of processing a watchpoint. This means our implementation of watchpoints is sensitive to the false positive rate, and we could expect watchpoints that share memory pages with popular variables to slow replay significantly.

Fortunately, the same data suggests that executing the user commands attached to a watchpoint is inexpensive. Reading or writing variables or calling a safe function adds less than a millisecond of latency over a null command, which is only a few milliseconds slower than a false positive. The safe function call is slightly slower than simple variable access, presumably due to the extra work by GDB to set up a temporary stack, marshal data, and clean up afterward.

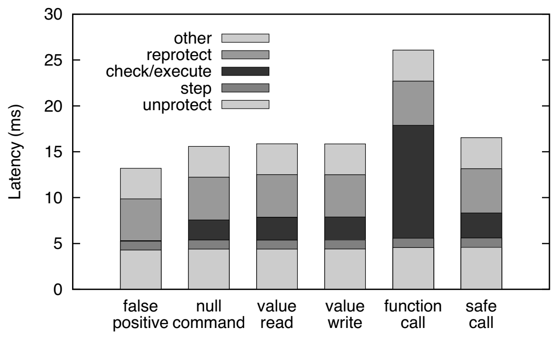


Figure 3: Latency breakdown for various watchpoint events.

A normal "unsafe" function call, on the other hand, is 50% slower than a safe one. The difference (9.6 ms) is attributed directly to the cost of temporarily disabling the watchpoint before invoking the function.

We break down the processing latency into phases:

- *Unprotect*: Temporarily disable memory protection on the watched variable's page, so that the faulting instruction can complete. This step requires calling `mprotect` for the application, through GDB.
- *Step*: Re-execute the faulting instruction. This requires a temporary breakpoint, used to return to the instruction from the fault handler.
- *Reprotect*: Re-enable protection with `mprotect`.
- *Check and Execute*: If the faulting address falls in a watched variable (as opposed to a false positive), its new value is extracted from GDB. If the value has changed, any attached command is evaluated by the Python interpreter.
- *Other*: Miscellaneous tasks, including reading the faulting address from the signal's user context.

Figure 3 shows that a false positive costs the same as a watchpoint hit. The dark segments in the middle of each bar show the portion required to execute the user command. It is small except the unsafe function call, where it dominates.

5.2 Micro-benchmarks: Scaling of Commands

Next we explored the scaling behavior of the four command micro-benchmarks: *value read*, *value write*, *function call*, and *safe call*. Figure 4 shows the cost of processing a watchpoint as the command accesses an increasing number of nodes. Each data point is averaged over the same number of watchpoints; the latency increases because more GDB instances must be contacted.

The figure includes the best-fit slope for each curve, which approximates the overhead added for each additional node that the command reads, writes, or calls. For most of the curves this amount closely matches the difference between a null command and the correspond-

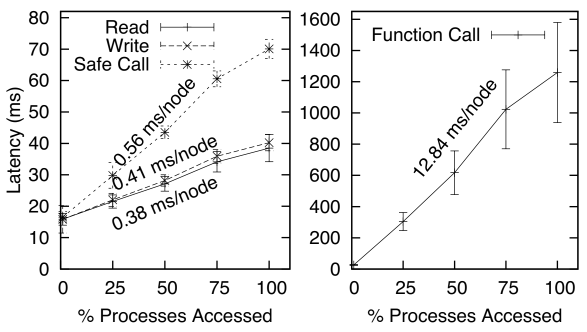


Figure 4: Micro-benchmarks indicating latency and first standard deviation (y axis), as a function of the percentage of nodes involved in the operation (x axis). The population contains 62 nodes.

ing single-node reference. In contrast, the unsafe function call benchmark increases at a faster rate—almost double—and with higher variance than predicted by the single node overhead. We attribute both phenomena to greater contention in the replay host's memory hierarchy due to the extra memory protection operations.

5.3 Micro-benchmarks on Chord

We continue by evaluating how the same primitive operations described in the previous section affect a baseline replay of a distributed application. For each benchmark, we average across 6 consecutive minute-long periods from the i3/Chord overlay logs described above. Other applications would experience more or less overhead, depending on the relative frequency of `libc` calls and watchpoint triggers.

We establish a replay baseline by replaying all 62 traced nodes in `liblog` without additional debugging tasks. Average replay slowdown is $3.12x$, with a standard deviation of $.08x$ over the 6 samples. `liblog` achieves a slowdown less than the expected $62x$ by skipping idle periods in each process. For comparison, simply replaying the logs in GDB, but without `liblog`, ran 11 times faster, for a replay *speedup* of $3.5x$. The difference between GDB and `liblog` is due to the scheduling overhead required to keep the 62 processes replaying consistently. `liblog` must continually stop the running process, check its progress, and swap in a new process to keep their virtual clocks synchronized. Without `liblog`, we let GDB replay each log fully before moving on.

To measure false positives, we add a dummy watchpoint on a variable at a memory page written about 4.7 times per second per replayed node; the total average replay slowdown goes up to $7.95x$ ($0.2x$ standard deviation), or $2.55x$ slower than baseline replay. This is greater than what our micro-benchmarks predict: 4.7 triggered watchpoints per second should expand every replayed second from the baseline 3.12 seconds by an additional $4.7 \times 62 \times 0.0132 = 3.87$ seconds for a slowdown of $4.87x$. We conjecture that the extra slowdown is due to cache contention on the replay machine, though further testing will be required to validate this.

Benchmark	Slowdown	(dev)	Relative
No Watchpoints	3.12	(.08)	1
False Positives Only	7.95	(0.22)	2.55
Null Command	8.24	(0.24)	2.64
Value Read	8.25	(0.17)	2.65
Value Write	8.26	(0.21)	2.65
Function Call	9.01	(0.27)	2.89
Safe Call	8.45	(0.26)	2.71

Table 2: Micro-benchmarks: slowdown of Chord replay for watchpoints with different commands.

To measure `Friday`'s slowdown for the various types of watchpoint commands, we set a watchpoint on a variable that is modified once a second on each node. This watchpoint falls on the same memory page as in the previous experiment, so we now see one watchpoint hit and 3.7 false positives per second. The slowdown for each type of command is listed in Table 2.

The same basic trends from the micro-benchmarks appear here: function calls are more expensive than other commands, which are only slightly slower than null commands. Significantly, the relative cost of the commands is dwarfed by the cost of handling false positives. This is expected, because the latency of processing a false positive is almost as large as a watchpoint hit, and because the number of false positives is much greater than the number of hits for this experiment. We examine different workloads later, in Section 5.4.

Next, we scale the number of replayed nodes on whose state we place watchpoints, to verify that replay performance scales with the number of watchpoints. These experiments complement the earlier set which verified the scalability of the commands.

As expected, as the number of memory pages incurring false positives grows, replay slows down. Figure 5(a) shows that the rate at which watchpoints are crossed—both hits and false positives—increases as more processes enable watchpoints. The correlation is not perfect, because some nodes were more active and executed the watched inner loop more often than others.

Figure 5(b) plots the relative slowdown caused by the different types of commands as the watchpoint rate increases. These lines suggest that `Friday` does indeed scale with the number of watchpoints enabled and false positives triggered.

5.4 Case Studies

Finally, we return to the case studies from Section 4. Unlike the micro-benchmarks, these case studies include realistic and useful commands. They exhibit a range of performance, and two of them employ distributed break-

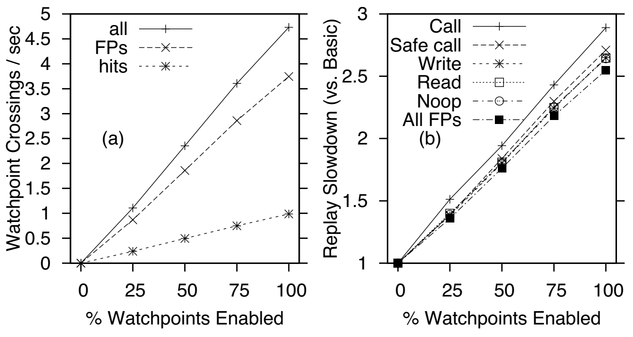


Figure 5: (a) Number of watchpoints crossed vs. percentage of nodes with watchpoints enabled (i3/Chord logs). Approximately linear. (b) Replay slowdown vs. percentage of nodes with watchpoints enabled, relative to baseline replay (i3/Chord logs).

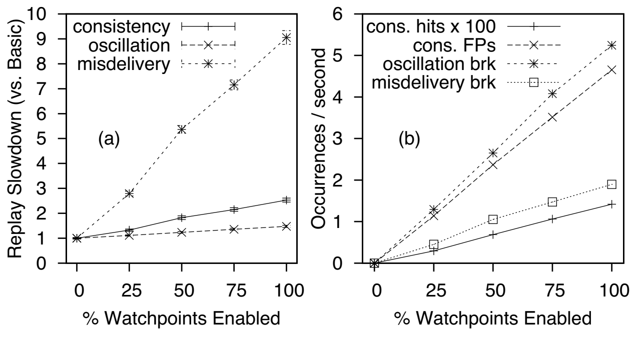


Figure 6: (a) Replay slowdown statistics for case study predicate performance vs. percentage of nodes with watchpoints enabled. (b) Watchpoint, breakpoint, and false positive rates vs. percentage of nodes with watchpoints/breakpoints enabled.

Predicate	Slowdown
None	1.00
Ring Consistency Stat.	2.53
w/Software Watchpoints	8470.0
State Oscillation	1.48
Misdelivered Packets	9.05

Table 3: Normalized replay slowdown under three different case studies. The last row gives the slowdown for the Ring Consistency Statistics predicate when implemented in GDB with single-stepping.

points instead of watchpoints.

We replayed the same logs used in earlier experiments with the predicates for Misdelivered Packets (Section 4.1.2), Ring Consistency Statistics (Section 4.1.4), and State Oscillation (Section 4.1.5). Figure 6 plots the relative replay speed against the percentage of nodes on which the predicates are enabled. Table 3 summarizes the results. Results with the case studies from Section 4.2 were comparable, giving a 100%-coverage slowdown of about 14 with a population of 10 nodes.

Looking at the table first, we see that the three case studies range from 1.5 to 9 times slower than baseline replay. For comparison, we modified `Friday` to use software watchpoints in GDB instead of our memory protection-based system, and reran the Ring Consistency Statistics predicate. As the table shows, that experiment took over 8000 times longer than basic replay, or about 3000 times slower than `Friday`'s watchpoints. GDB's software watchpoints are implemented by single-stepping through the execution, which consumes thousands of instructions per step. The individual memory protection operations used by `Friday` are even more expensive but their cost can be amortized across thousands of non-faulting instructions.

Turning to Figure 6, the performance of the Ring Consistency Statistics predicate closely matches that of the micro-benchmarks in the previous section (cf., Figure 5(b)). This fact is not surprising: performance here is dominated by the false positive rate, because these predicates perform little computation when triggered. Furthermore, both sets of predicates watch variables located on the same page of memory, due to the internal structure of the i3/Chord application, so their false positive rates are the same.

The figure shows that the State Oscillation predicate encounters more breakpoints than the Ring Consistency predicate does watchpoints. However, handling a breakpoint is almost free, and the commands are similar in complexity, so `Friday` runs much faster for State Oscillation predicates.

The Misdelivered Packets case study hit even fewer breakpoints, and ran the fewest number of commands. Those commands were very resource-intensive, however, requiring dozens of (safe) function calls each time. Overall performance, as shown in Figure 6(a), is the slowest of the three predicates.

6 Related Work

`Friday` utilizes library interposition to obtain a replayable deterministic trace of distributed executions. The WiDS Checker [17] has many similar characteristics with some notable differences: whereas `Friday` operates on unmodified applications and checks predicates at single-machine-instruction granularity, the WiDS Checker is applicable only to applications developed with the WiDS toolkit and checks predicates at event-handler granularity. Similarly to `Friday`, Jockey [20] and Flashback [23] use system call interposition, binary rewriting, and operating system modifications to capture deterministic replayable traces, but only for a single node. DejaVu [15] targets distributed Java applications, but lacks the state manipulation facilities of `Friday`.

Further afield, much research has gone into replay debugging via virtualization, which can capture system effects below the system library level, first articulated by Harris [9]. Several projects have pursued that agenda since [11, 13, 23], albeit only for single-thread, single-process, or single-machine applications. Furthermore, symbolic debugging in such systems faces greater challenges than with Friday, since the "semantic gap" between application-defined symbols and the virtual machine interface must be bridged at some computational and complexity cost.

Moving away from replay debugging, many systems focus on extracting execution logs and then mining those logs for debugging purposes [1, 2, 5, 6, 10, 22]. Such systems face the challenge of reconstructing meaningful data- and control-flow from low-level logged monitoring information. Friday circumvents this challenge, since it can fully inspect the internal state of the nodes in the system during a replay of the traced execution and, as a result, need not guess at connections across layers (as with black-box approaches) or recompile the system (as with annotation-based systems).

Notable logging-based work in closer alignment with Friday comes from the Bi-directional, Distributed BackTracker (BDB) [14], XTrace [7], and Pip [19]. BDB and XTrace track and report causality among events within a distributed system, e.g., to trace identified back-door programs backwards to their onset or, in the case of XTrace, to identify problems along cross-layer paths. Pip [19] works by comparing actual behavior and expected behavior to expose bugs. Such behaviors are defined as orderings of logged operations at participating threads and limits on the values of annotated and logged performance metrics. In both cases, the kinds of checks performed can be readily encoded in Friday, except for those dependent on kernel-level sensors, which lie beyond our library tracing granularity. However, the replay-based nature of Friday allows programmers to refine checks after repeated replays without the need for recompilation and fresh log extraction, as would be the case for disambiguating noisy tasks (e.g., directory listing filesystem operations in BDB) or for creating new sensors (e.g., heap size monitors when none were initially thought necessary in Pip).

At a more abstract level, model checking has been recently proposed as a tool for debugging distributed systems. Most notably, MaceMC [12] is a heuristic model checker for finding liveness violations in distributed applications built using the Mace language. As with model checking in general, MaceMC can exercise a distributed application over many more possible executions than any replay debugging system, include Friday, can. However replay systems, such as Friday, tend to capture more realistic problems than model checkers such as complex network failures and hardware malfunctions, and can typically operate on much longer actual executions than the combinatorial nature of model checking can permit.

A growing body of work is starting to look at on-line debugging [27], in contrast to the off-line nature of debuggers described above. The P2 debugger [21] operates on the P2 [18] system for the high-level specification and implementation of distributed systems. Like Friday, this debugger allows programmers to express distributed invariants in the same terms as the running system, albeit at a much higher-level of abstraction than Friday's libc-level granularity. Unlike Friday, P2 targets on-line invariant checking, not replay execution. As a result, though the P2 debugger can operate in a completely distributed fashion and without need for log back-hauling, it can primarily check invariants that have efficient on-line, distributed implementations. Friday, however, can check expensive invariants such as the existence of disjoint paths, since it has the luxury of operating outside the normal execution of the system.

More broadly, many distributed monitoring systems can perform debugging functions, typically with a statistical bend [4,28,30]. Such systems employ distributed data organization and indexing to perform efficient distributed queries on the running system state, but do not capture control path information equivalent to that captured by Friday.

7 Conclusion and Future Work

Friday is a replay-based symbolic debugger for distributed applications that enables the developer to maintain global, comprehensive views of the system state. It extends the GDB debugger and liblog replay library with distributed watchpoints, distributed breakpoints, and actions on distributed state. Friday provides programmers with sophisticated facilities for checking global invariants—such as routing consistency—on distributed executions. We have described the design, implementation, usage cases, and performance evaluation for Friday, showing it to be powerful and efficient for distributed debugging tasks that were, thus far, underserved by commercial or research debugging tools.

The road ahead is ripe for further innovation in distributed debugging. One direction of future work revolves around reducing watchpoint overheads via the reimplementation of the malloc library call and memory page fragmentation, or through intermediate binary representations, such as those provided by the Valgrind tool. Building a hybrid system that leverages the limited hardware watchpoints, yet gracefully degrades to slower methods, would also be rewarding.

Another high-priority feature is the ability to checkpoint Friday state during replay. This would allow a programmer to replay in Friday a traced session with its predicates from its beginning, constructing any debugging state along the way, but only restarting further debugging runs from intermediate checkpoints, without the need for reconstruction of debugging state.

We are considering better support for thread-level parallelism in `Friday` and `liblog`. Currently threads execute serially with a cooperative threading model, to order operations on shared memory. We have also designed a mechanism that supports preemptive scheduling in userland, and we are also exploring techniques for allowing full parallelism in controlled situations.

We plan to expand our proof-of-concept cluster-replay mechanism to make more efficient use of the cluster's resources. Our replay method was designed to ensure that each replay process is effectively independent and requires little external communication. Beyond cluster-parallelism, we are developing a version of `liblog` that allows replay in-situ on PlanetLab. This technique increases the cost of centralized scheduling but avoids the transfer of potentially large checkpoints and logs.

Further down the road, we want to improve the ability of the system operator to reason about time. Perhaps our virtual clocks could be optimized to track "real" or *average* time more closely when the distributed clocks are poorly synchronized. Better yet, it could be helpful to make stronger statements in the face of concurrency and race conditions. For example, could `Friday` guarantee that an invariant *always* held for an execution, given all possible interleavings of concurrent events?

Growing in scope, `Friday` motivates a renewed look at on-line distributed debugging as well. Our prior experience with P2 debugging [21] indicates that a higher-level specification of invariants, e.g., at "pseudo-code level," might be beneficially combined with system library-level implementation of those invariants, as exemplified by `Friday`, for high expressibility yet deep understanding of the low-level execution state of a system.

Acknowledgments: We are indebted to Sriram Sankararaman for allowing us the use of his Tk implementation, to the anonymous reviewers for their astute feedback, and to Mike Dahlin for his shepherding efforts. This research was sponsored by NSF under grant number ANI-0133811, by the State of California under MICRO grants #05-060 and #06-150, and by a Fannie and John Hertz Foundation Graduate Fellowship.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *SOSP*, 2003.

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *OSDI*, 2004.

[3] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating system support for planetary-scale network services. In *NSDI*, 2004.

[4] A. R. Bharambe, M. Agrawal, and S. Seshan. Mercury: supporting scalable multi-attribute range queries. In *SIGCOMM*, 2004.

[5] A. Chanda, K. Elmeleegy, A. Cox, and W. Zwaenepoel. Causeway: System Support for Controlling and Analyzing the Execution of Distributed Programs. In *HotOS*, 2005.

[6] M. Y. Chen, A. Accardi, E. Kıcıman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based Failure and Evolution Management. In *NSDI*, 2004.

[7] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica. XTrace: A Pervasive Network Tracing Framework. In *NSDI*, 2007.

[8] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay Debugging for Distributed Applications. In *USENIX Annual Technical Conference*, 2006.

[9] T. L. Harris. Dependable Software Needs Pervasive Debugging (Extended Abstract). In *SIGOPS EW*, 2002.

[10] J. Hollingsworth and B. Miller. Dynamic Control of Performance Monitoring of Large Scale Parallel Systems. In *Super Computing*, 1993.

[11] A. Joshi, S. T. King, G. W. Dunlap, and P. M. Chen. Detecting Past and Present Intrusions through VulnerabilitySpecific Predicates. In *SOSP*, 2005.

[12] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI*, 2007.

[13] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *USENIX Annual Technical Conference*, 2005.

[14] S. T. King, Z. M. Mao, D. G. Lucchetti, and P. M. Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[15] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *IPDPS*, 2000.

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978.

[17] X. Liu, W. Lin, A. Pan, and Z. Zhang. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*, 2007.

[18] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *SOSP*, 2005.

[19] P. Reynolds, J. L. Wiener, J. C. Mogul, M. A. Shah, C. Killian, and A. Vahdat. Pip: Detecting the Unexpected in Distributed Systems. In *NSDI*, 2006.

[20] Y. Saito. Jockey: A user-space library for record-replay debugging. In *International Symposium on Automated Analysis-Driven Debugging*, 2005.

[21] A. Singh, P. Maniatis, T. Roscoe, and P. Drushel. Using Queries for Distributed Monitoring and Forensics. In *EuroSys*, 2006.

[22] R. Snodgrass. A Relations Approach to Monitoring Complex Systems. *IEEE Transactions on Computer Systems*, 6(2):157–196, 1988.

[23] S. M. Srinivashan, S. Kandula, C. R. Andrews, and Y. Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *USENIX Annual Technical Conference*, 2004.

[24] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *SIGCOMM*, 2002.

[25] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. *IEEE/ACM Transactions of Networking*, 11(1):17–32, 2003.

[26] L. Subramanian. *Decentralized Security Mechanisms for Routing Protocols*. PhD thesis, University of California at Berkeley, 2005.

[27] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Automatic On-line Failure Diagnosis at the End-User Site. In *HotDep*, 2006.

[28] R. van Renesse, K. P. Birman, D. Dumitriu, and W. Vogel. Scalable management and data mining using Astrolabe. In *IPTPS*, 2002.

[29] R. Wahbe. Efficient data breakpoints. In *ASPLOS*, 1992.

[30] P. Yalagandula and M. Dahlin. A Scalable Distributed Information Management System. In *SIGCOMM*, 2004.

Network Coordinates in the Wild

Jonathan Ledlie, Paul Gardner, and Margo Seltzer
Harvard School of Engineering and Applied Sciences and Aelitis
{jonathan,margo}@eecs.harvard.edu, pgardner@aelitis.com

Abstract

Network coordinates provide a mechanism for selecting and placing servers efficiently in a large distributed system. This approach works well as long as the coordinates continue to accurately reflect network topology. We conducted a long-term study of a subset of a million-plus node coordinate system and found that it exhibited some of the problems for which network coordinates are frequently criticized, for example, inaccuracy and fragility in the presence of violations of the triangle inequality. Fortunately, we show that several simple techniques remedy many of these problems. Using the Azureus BitTorrent network as our testbed, we show that live, large-scale network coordinate systems behave differently than their tame PlanetLab and simulation-based counterparts. We find higher relative errors, more triangle inequality violations, and higher churn. We present and evaluate a number of techniques that, when applied to Azureus, efficiently produce accurate and stable network coordinates.

1 Introduction

The performance of many Internet applications, such as distributed hash tables, web caches, and overlay networks, relies on accurate latency estimation between participants (*e.g.,* [10, 27]). Researchers propose acquiring these measurements using various techniques, from proxy measurement [13, 26] to landmark binning [23] to decentralized network embeddings (*e.g.,* [11, 19, 21, 24, 28]). In a network embedding, a subset of inter-node latency measurements is embedded into a low-dimensional metric space. Each node maintains a *network coordinate*, such that the metric distance between two coordinates in the abstract space predicts real-world latencies. This paper examines the performance of Internet-scale network embeddings through the study of a subset of a million-node live coordinate system.

Although network coordinates have attractive properties for latency prediction on the Internet, they have been criticized for requiring expensive maintenance and having prediction accuracy significantly worse than direct measurement methods such as Meridian [29]. At the very least, critics say that network coordinates are an unproven idea and unlikely to work in practice because Internet routing policies cause too many triangle inequality violations [31]. Supporters respond with claims that accuracies are reasonable ($8-15\%$), and they have demonstrated that coordinate maintenance can be built on top of existing application communication. They support these claims with simulations and small-scale live deployments on PlanetLab [8, 16, 20, 24].

This paper provides the missing piece of the debate: data and analysis of a truly large-scale and long-running network coordinate system. The Azureus file-sharing network [1], which runs a million-node network coordinate system, is the main artifact for our analysis and experimentation. This work is the result of a collaboration between the Azureus team (Gardner) and a team from Harvard (Ledlie, Seltzer). Gardner contacted the Harvard team because Azureus was exhibiting some of the difficulties that Ledlie *et al.* had addressed in earlier work with a PlanetLab-based coordinate system [16]. We merged the techniques from Ledlie's previous work into the test branch of the Azureus code, used by approximately ten thousand clients.

While our previous techniques did work "in the wild," Azureus continued to experience unsatisfactorily high errors. This occurred because its gossip pattern stifled convergence: as all coordinate maintenance is "piggybacked" on other traffic, each coordinate became heavily skewed to small segments of the network and failed to become globally accurate. We created a simple new technique called *neighbor decay* that smoothly manages these skewed neighbor sets while retaining the appealing zero-maintenance property of Azureus' coordinates. With these techniques in place, Azureus' coordinates and, by inference, Internet-scale coordinate systems in general, can now tackle a basic goal: quickly and efficiently optimizing anycast decisions based on correct latency estimates. Because even with these approaches Internet-scale coordinates are still partially untamed, we isolated and analyzed a set of major remaining impediments.

The contributions of this work are:

- Improvements to the live Azureus coordinate system, producing a 43% improvement in accuracy and a four order-of-magnitude improvement in stability. The new coordinates optimize DHT traversal, helping the application pick physically close nodes; this trims lookup delay by 33% compared to the most direct logical path.

- A new technique for managing neighbors in coordinate systems where all gossip is "piggybacked" on existing traffic — *i.e.* where there are zero maintenance messages.
- A new, large-scale latency matrix providing a valuable new portal into Internet behavior. Previous large matrices were between DNS servers and did not capture latencies between actual nodes [8, 29].
- Evidence *why* Internet-scale latency estimation with coordinates works. We find the intrinsic dimensionality of large-scale systems to be *less* than previous work, which studied smaller networks [28], and we show *why* the world flattens into near-planar Euclidean coordinates.
- Analysis of five major barriers to accuracy: churn, drift, intrinsic error, corruption, and latency variance. We present techniques for lowering these barriers and show how latency variance requires a fundamentally new approach to latency prediction.

In Section 2, we explain why practitioners, such as the Azureus developers, use network coordinates in large-scale deployments and review Azureus' network coordinate algorithm. In Section 3, we use a dense latency matrix to analyze the characteristics of the Azureus' latency distribution, determining its intrinsic dimensionality and the extent of its triangle inequality violations. In Section 4, we describe three techniques integrated into the Azureus code. In Section 5 we review metrics for evaluating coordinate systems. In Section 6, we examine the live performance of Azureus through three methods: (a) Azureus clients we ran on PlanetLab, (b) crawling instrumented clients run by approximately ten thousand Azureus users, and (c) an application-level benchmark: using coordinates to optimize DHT hop selection. In Section 7, we examine five primary causes of the remaining difference between the current live accuracy and what appears to be achievable based on simulation results. In Section 8, we review the approaches for estimating latencies in large distributed systems. In Section 9, we conclude.

2 Background

Azureus is currently one of the most popular clients for BitTorrent, a file sharing protocol [6]. For a given file, the protocol embodies four main roles: an *initial seeder*, *new seeders*, a *tracker*, and *peers*. Initial seeders, new seeders, and peers are all transient *clients*; trackers are typically web servers. The initial seeder is the source of the file. It divides the file into small pieces, creates a metadata description of the file and sends this description to the tracker. Peers discover this file description through some out-of-band mechanism (*e.g.,* a web page) and then begin looking for pieces of the file. Peers contact the tracker to bootstrap their knowledge of other peers and seeds. The tracker returns a randomized subsets of other peers and seeds. Initially, only the initial seeder has pieces, but soon peers are able to exchange missing pieces with each other, typically using a tit-for-tat scheme. Once a peer acquires all of the pieces for a file, it becomes a new seeder. This collection of clients actively sharing a file is called a *swarm*. In Azureus, file descriptors and other metadata are stored in a DHT, in which all clients participate, and any node can be assigned the role of tracker if it is or is near the root of the hash of a given file's descriptor. In practice, there can be many possible trackers from which to choose for a particular file and even more possible clients for a given piece. A key challenge in efficiently implementing this protocol is providing a simple method for node selection, an example of *anycast*.

Distributed systems developers are beginning to use network coordinates as a mechanism to support anycast. The Azureus developers use them for two distinct purposes: (a) to optimize DHT traversal and (b) to select nearby nodes for application-level congestion monitoring. We are currently testing another coordinate-based optimization: biasing the set of nodes the tracker returns to be nearby the caller. Bindal *et al.* show in simulation how these locally-biased swarms reduce download times and inter-ISP traffic [3]. Future plans call for using network coordinates to optimize media streaming over Azureus.

We worked with the Azureus developers to analyze and improve the coordinates maintained by their system, which contains more than a million clients. We were able to modify the Azureus code internals and watch its behavior on a subset of the network because approximately ten thousand Azureus users run a plugin that automatically upgrades their version to the latest CVS release. According to the Azureus developers, the clients who use the latest release exhibit normal user characteristics, so we expect that our results generalize to the larger system.

2.1 Vivaldi

Azureus uses the Vivaldi network coordinate update algorithm [8]. The *Vivaldi* algorithm calculates coordinates as the solution to a spring relaxation problem. The measured latencies between nodes are modeled as the extensions of springs between massless bodies. A network embedding with a minimum error is found as the low-energy state of the spring system.

Figure 1 shows how a new observation, consisting of a remote node's coordinate $\vec{x_j}$, its confidence w_j, and a latency measurement l_{ij} between the two nodes, i and j, is used to update a local coordinate. The *confidence*, w_i, quantifies how accurate a coordinate is believed to be. Note that confidence increases as it approaches 0. The al-

```
VIVALDI($\overrightarrow{x_j}, w_j, l_{ij}$)
1  $w_s = \frac{w_i}{w_i + w_j}$
2  $\epsilon = \frac{|\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}|}{l_{ij}}$
3  $\alpha = c_e \times w_s$
4  $w_i = (\alpha \times \epsilon) + ((1 - \alpha) \times w_i)$
5  $\delta = c_c \times w_s$
6  $\overrightarrow{x_i} = \overrightarrow{x_i} + \delta \times (\|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}) \times u(\overrightarrow{x_i} - \overrightarrow{x_j})$
```

Figure 1: *Vivaldi* update algorithm.

gorithm first calculates the *sample confidence* w_s (Line 1) and the *relative error* ϵ (Line 2). The relative error ϵ expresses the accuracy of the coordinate in comparison to the true network latency. Second, node i updates its confidence w_i with an exponentially-weighted moving average (EWMA) (Line 4). The weight α for the EWMA is set according to the sample confidence w_s (Line 3). Also based on the sample confidence, δ dampens the change applied to the coordinate (Line 5). As a final step, the coordinate is updated in Line 6 (u is the unit vector). Constants c_e and c_c affect the maximum impact an observation can have on the confidence and the coordinate, respectively.

Height is an alternative to a purely Euclidean distance metric. With *height*, the distance between nodes is measured as their Euclidean distance plus a *height* "above" the hypercube that models the latency penalty of network access links, such as DSL lines [8].

Each node successively refines its coordinate through periodic updates with other nodes in its *neighbor set*. In Azureus, the information used to maintain the network coordinate system is entirely piggybacked on existing messages, such as routing table heartbeats. While this does mean the coordinates induce no additional overhead (beyond 24 bytes per message for four dimensions, height, and confidence), it also means that the algorithm needed to be modified to function *passively*. In Section 4.2, we describe a technique we developed to incorporate information from highly transient neighbors.

3 Latencies in the Wild

Before we examine the accuracy with which Internet-scale latencies can be embedded into a coordinate space, we compare latencies in Azureus to those in other networks to gain insight into the causes of error in Internet-scale embeddings. We generate a dense latency matrix of a subset of Azureus and compare it to PlanetLab and to the MIT King data set, a square matrix containing the median latencies between 1740 DNS servers collected using the King method [8, 13]. Researchers found PlanetLab and MIT King can be reduced to low dimensional coordinates with $\leq 10\%$ median error [8, 16]. We examine three characteristics: inter-node round trip times, violations of the triangle inequality, and intrinsic dimensionality.

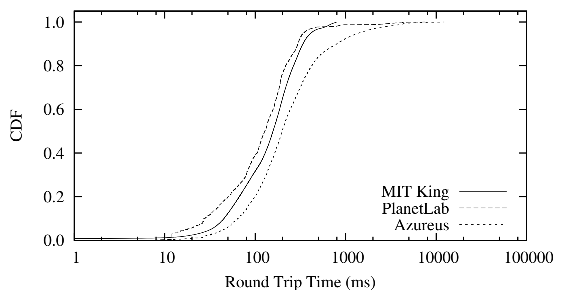


Figure 2: A comparison of round-trip times shows that Azureus spreads across a range one order-of-magnitude larger than MIT King, based on inter-DNS latencies. This larger spread tends to lead to lower accuracy embeddings.

3.1 Collection

We instrumented clients that we ran on PlanetLab to record the application-level latency between them and the rest of the network creating a dense latency matrix. These clients ran on 283 PlanetLab nodes for 24 days starting on July 19^{th} 2006, collecting 9.5×10^7 latency measurements to $156,658$ Azureus nodes. To reduce these raw measurements into a dense latency matrix, we used the following process: first, we summarized each edge with the median round trip time for this edge, discarding edges with fewer than a minimum number of samples (4); second, we discarded all nodes that had fewer than half of the maximum number of edges (280). This process resulted in a 249×2902 matrix with 91% density, where 83% of the entries were the median of at least ten samples. We derived the PlanetLab data set from the Azureus matrix by simply selecting out its subset of hosts.

3.2 Round Trip Times

In Figure 2, we illustrate the distribution of inter-node round trip times between nodes in the three data sets. The King measurements were limited to a maximum of $800ms$. The data exhibit one important characteristic: spread. The application-level, Azureus round trip times spread across four orders-of-magnitude, while the inter-DNS, King data set spreads across three. In theory, this is not a harbinger of higher embedding error; in practice, however, as Hong *et al.* have shown, the error between nodes whose distance is near the middle of the latency distribution tends to be the lowest [30]: with longer tails to this distribution, there are more edges to be inaccurate. (We found ICMP measurements exhibit a similarly wide distribution; see § 7.5.) This wide spread is a warning sign that Azureus will have higher error than a system with a narrower round trip time distribution.

3.3 Violations of the Triangle Inequality

Network coordinate embeddings that use Euclidean distances make the assumption that the triangle inequality is

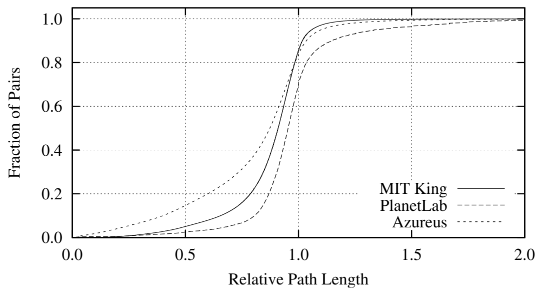


Figure 3: In all three data sets, over half of all node pairs fail the Tang/Crovella triangle inequality test, because there exists a third node between the nodes in the pair that produces a shorter path than the direct path between the two nodes. A large fraction of these violating pairs have paths that are significantly faster.

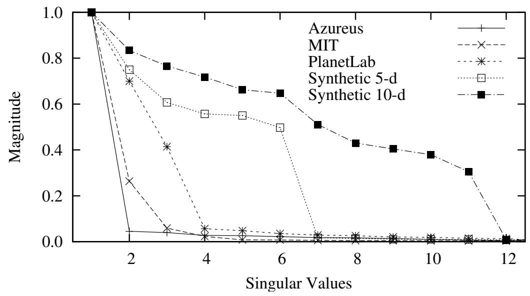


Figure 4: Scree plots suggest the inherent dimensionality of MIT King, PlanetLab, and Azureus datasets is small. Two synthetic matrices of five and ten dimensions are included for comparison.

not violated to a great extent by a large fraction of pairs of nodes. The triangle inequality states that for any triangle the length of a given side must be less than the sum of the other two sides but greater than the difference between the two sides, *i.e.,* the sides must be able to form a triangle. When the latencies between node triples cannot form a triangle, they are said to violate the triangle inequality. Nodes with large and frequent violations tend to be the ones with the largest individual prediction error and their existence decreases overall accuracy (see [16] and § 7.3).

We use a method from Tang and Crovella to examine the severity of triangle inequality violations [28]. This method normalizes the severity of each violation, permitting an all-pairs comparison. For each node pair, we find the shortest path between the two that passes through a third node. Thus, for all pairs of nodes i and j, we find the best alternative path through a node k and normalize by the latency between i and j:

$$rpl = min_k \left(\frac{d(i,k) + d(k,j)}{d(i,j)} \right)$$

Figure 3 illustrates the cumulative distribution of this quantity, the relative path length. Note that any fraction below 1 is a violation: there exists a path through an alternative node that is faster than the direct path. 83% of the Azureus pairs, 85% of MIT King, and 68% of the PlanetLab subset violate the triangle inequality. In contrast to earlier work that examined several small-scale data sets [28], we find the fraction of pairs with the largest violations to be quite large: Tang and Crovella found only 10% of nodes had an alternative path that is $\geq 20\%$ faster; here 37% of Azureus pairs and 22% of MIT King pairs exhibit this large level of violation.

We examined the cause of the large fraction of pairs with very low *rpl* (< 0.1) in Azureus. We found that only a few nodes were members of many of these low *rpl* pairs. What distinguished these nodes — and what was the cause of their frequent participation in triangle inequality violations — was that their delay to non-PlanetLab nodes was atypically large, on the order of seconds, while their delay to other PlanetLab nodes remained typical (less than a second). In effect, this extended one side of the triangles these nodes participated in: $d(i,j)$ became large while $d(i,k)$ and $d(k,j)$ did not. Because PlanetLab nodes that exhibited this behavior were co-located, we conjecture that the Azureus traffic to non-PlanetLab sites was being artificially limited at site gateways, while traffic to PlanetLab nodes avoided this traffic shaping. Rather than being a construct of the PlanetLab environment, this effect, leading to bi- or multi-modal latency distributions, will be the norm for at least some participants in Internet-scale applications that use well-known ports and consume a large amount of bandwidth, such as Azureus, because some sites will limit traffic and some will not. Like the round trip time spread, Azureus' violations foreshadow a higher embedding error.

3.4 Dimensionality

Network coordinates would be less useful if a large number of dimensions were needed to capture the inter-node latencies of the Internet. Tang and Crovella used Principal Component Analysis (PCA) to hint at the number of dimensions required to encompass this information for several small data sets [28]. Because we wanted to know if few dimensions would be sufficient for a large, broad spectrum of endpoints, we used the same method to examine the intrinsic dimensionality of Azureus.

PCA is a linear transformation from one coordinate system to a new, orthogonal coordinate system. The new system is chosen such that each subsequent axis captures the maximum possible remaining variance in projections from points in the old system to points in the new: the first new axis captures the most variance, the second less, and so on. While an input system of k elements will produce an output system also of k elements, often only the first several dimensions of the output system will summarize all or part of the same distance information of the original set

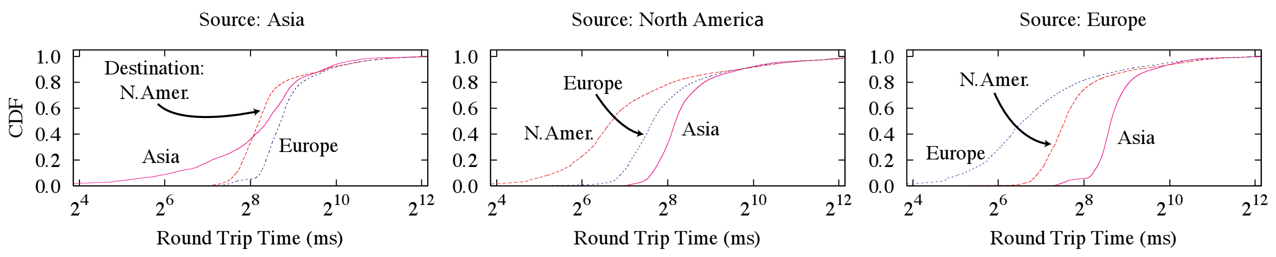


Figure 5: Intercontinental Latency Distributions illustrate why a Euclidean distance metric works for network coordinates on the Internet: messages from Asia to Europe (and from Europe to Asia) go through North America.

of points. Singular values are a result of the PCA transformation: each new axis has a corresponding singular value that describes the amount of variance captured by this axis. Thus, if a singular value is very small or zero, this suggests that this axis is unnecessary in describing the variance in a particular data set.

Because PCA requires a full matrix, we first used the following two techniques to fill in the remaining 9% of the Azureus matrix and the missing 0.4% of the MIT matrix. We filled half of the missing Azureus values with the King technique [13] (King fails in certain cases, *e.g.,* when the endpoint cannot be resolved). We interpolated the remaining values in both matrices by embedding each matrix and extracting the missing values.

We use a scree plot to illustrate how much variance each new singular value is capturing, which in turn hints at the inherent dimensionality of the underlying data set. The independent variables of a scree plot are the singular values, sorted by their magnitude; the dependent variables are their corresponding magnitudes. At the point where the magnitude of the singular values becomes zero or nearly zero, the relative importance of this and subsequent singular values (*i.e.,* dimensions) is low. Up to this point, these dimensions are necessary to capture the values in the original input matrix, which in this case is made up of inter-node latency values.

We show the normalized singular values for the King, PlanetLab, and Azureus data sets in Figure 4. For comparison, we created synthetic $5d$ and $10d$ systems each containing 250 random points in a unit hypercube and found their singular values. As one would expect, the synthetic $5d$ and $10d$ data sets show a sharp knee soon after 5 and 10 singular values, respectively. In contrast, the bulk of the inter-node latency information from two Internet-based data sets requires very few dimensions. Azureus, in particular, is dominated by a single dimension, and MIT King by two. However, the next several dimensions remain significant for the few nodes that need to navigate around the clusters of nodes that have found good positions. In the data, this is shown by the continued relevance of singular values when compared to synthetic data sets. To lower the error for these nodes, we find $4-5$ dimensions is appropriate for Internet-scale network coordinates. While the previous two characteristics, round trip times and violations of the triangle inequality, suggest that the Azureus latency distribution will experience higher error than MIT King, its intrinsic dimensionality does not appear to be an additional impediment.

3.5 Intercontinental Latency Distributions

While the Azureus data set is clearly of low dimensionality, a more concrete way to examine the "flatness" of this large-scale network is to look at its intercontinental latency distribution. In a way, it is surprising that embedding latencies found on a globe (the Earth) into a Euclidean space works at all. If messages could be routed in any direction of the Earth's surface, using a Euclidean metric would be a poor choice. Previous work on spherical coordinates, however, found they had significantly larger error than Euclidean ones [8]. Anecdotal evidence suggested that the main reason why the Internet embeds into a low dimensional Euclidean space is because the world is flat: traffic between Asia and Europe flows through North America [8].

An examination of our Azureus data set confirms that this traffic flow is indeed the case. We mapped the IP addresses in the data set to countries through their autonomous system record and, in turn, mapped these countries to continents. As Figure 5 illustrates, *no* messages from Asia to Europe were faster than those from Asia to North America; the same holds in the other direction. All paths between Asia and Europe appear to travel in a line across two oceans. This trend continues until the speed of the connection to ISPs or other coarse delays begin to dominate.

This flatness suggests why hyperbolic coordinates [25] also work well: North America maps to the center of the hyperbolic space. Thus, because the distribution of latencies is "flat" – at least at a high level – using a Euclidean metric is sufficient. In the future, new direct transmission lines between Europe and Asia may change the Internet's shape, perhaps driving a shift to spherical coordinates.

4 Taming Live Coordinate Systems

From our experience tuning a network coordinate system on PlanetLab, we developed two techniques that lead to more stable and accurate coordinates on a small "live" system [16]. The Azureus and Harvard teams worked together to integrate these techniques into the Azureus code. After confirming that these techniques worked as expected, we found and resolved a new problem: skewed neighbor sets. This problem particularly disrupts large-scale, live coordinate systems like Azureus that rely solely on other application communication for maintenance (*i.e.* they have zero maintenance costs) and has has been suggested as a goal for coordinate systems [8]. Through experimentation with these techniques in simulation and periodic measurement of the live system, we arrived at coordinates that are not perfect, but are a satisfactory start. We include a review of the techniques we developed as part of our previous research (§ 4.1) and describe our new technique, *neighbor decay* (§ 4.2).

4.1 Latency and Update Filters

In previous work, we developed two simple filters that had distinct beneficial effects on a coordinate system running on PlanetLab [16]. The first type, which we call a *latency filter*, takes the stream of latency measurements from a remote node and turns these into an expected latency value. For a stream of measurements between nodes i and j, the goal of the latency filter is to summarize the measurements, providing a current and stable description of the expected latency between i and j. Two main considerations affect the value $Ex[rtt(i, j)]$. First, anomalous measurements, sometimes several orders-of-magnitude larger than the baseline, would appear in the stream of measurements. For example, we would measure a round-trip time of $1000ms$ when typical measurements were $200ms$. Although we were using application-level UDP measurements, we found these anomalies also occurred with ICMP. Second, the expected value could not be fixed at a single value. Due to congestion and BGP changes, the underlying latency between pairs of nodes changes. We found that using a simple, short, moving median worked as a latency filter compensating for both anomalous measurements and plateau shifts.

The second type of filter we developed on PlanetLab focuses on making coordinates more stable, not more accurate. These *update filters* tackle a problem shared across many types of applications that use network coordinates: discerning when a coordinate has changed "enough" to potentially necessitate an application-level reaction (*e.g.,* a service migration). In an early application we developed that used network coordinates [22], we found it was hard for the application to immediately determine if it should react to coordinate updates, which were occurring several times per minute. A single threshold ("react if moved more than 50ms") did not work for all nodes because the volume through which each coordinate moved was node-dependent. We developed a generic filtering technique to allow applications to easily determine when to update coordinates. Applications that find all updates useful can bypass the filters.

Update filters make the distinction between constantly evolving "system-level" coordinates and stable "application-level" coordinates, providing a barrier between these two: system-level coordinates fine tune the coordinate further with each measurement, while application-level coordinates change only when the underlying coordinate has undergone a significant migration to a new location relative to other coordinates. In our previous work, we examined several heuristics for distinguishing between a system-level coordinate that was moving around a single point (not requiring application-level notification) and one that had migrated to a new location (potentially requiring application activity). We found heuristics that compare windows of previous system-level coordinates to one another, especially those that augment this comparison with distances to other nodes in the system, perform well. Applications can tune how much these windows may differ before being notified.

4.2 Neighbor Decay

Researchers have posited that a network coordinate subsystem could become a useful component of numerous large-scale distributed applications, particularly if it could perform its job *passively*, that is, without generating any extra traffic. In our Azureus implementation, this passivity was forced upon us: we had no control over the selection of which nodes we gossipped with or when we gossipped with them, because the information necessary for a coordinate update was piggybacked on to other application-level messages, *e.g.,* DHT routing table maintenance. Due to this passivity and to churn, nodes did not have fixed sets of neighbors with which they could expect regular exchanges. In fact, nodes would frequently receive $1 - 3$ updates from a remote node as that node was being tested for entry into the routing table and then never hear from that node again. The net effect of these limited exchanges was that each node's "working set" was much smaller than the number of nodes with which it actually communicated. Nodes were having blips of communication with many nodes, but constant communication with few. The goal of *neighbor decay* is to expand the size of the working set, which in turn improves accuracy.

A standard, gossip-based coordinate update involves taking new information from a single remote node and optimizing the local coordinate with respect to that node. If some set of remote nodes is sampled at approximately the same frequency, a node's coordinate will become optimized with respect to these remote coordinates (which are in turn performing the same process with their neighbors). However, if some remote nodes are sampled at a far greater frequency than others, the local coordinate optimization process will become skewed toward these nodes. In the theoretical limit, the result would be the same, but in practice, these skewed updates – a problem that could be expected in any passive implementation – slow the global optimization process.

Our solution to the problem of skewed neighbor updates is simple. Instead of refining our coordinate with respect to the remote node from which we just received new information, we refine it with respect to all nodes from which we have recently received an update. To normalize the sum of the forces of this *recent neighbor set*, we scale the force of each neighbor by its age: older information receives less weight. This allows nodes that we hear from only a few times to have a lasting, smooth effect on our coordinate. Algorithmically, we set the effect of a neighbor j on the aggregate force $\overrightarrow{F}$ to be:

$$\overrightarrow{F} = \overrightarrow{F} + \overrightarrow{F_j} \times \frac{a_{max} - a_j}{\sum(a_{max} - a)}$$

where a_j is the age of our knowledge of j and a_{max} is the age of the oldest neighbor.

This use of an expanded neighbor set that decays slowly over time has two main benefits. First, because the force from each update is effectively sliced up and distributed over time, nodes' coordinates do not jump to locations where they have high error with respect to other members of the neighbor set. Second, by keeping track of recent, but not old, neighbors, *neighbor decay* acts to increase the effective size of the neighbor set, which in turn leads to higher global accuracy. In our implementation, nodes expired from the *recent neighbor set* after 30 minutes.

Note the distinct effects of *neighbor decay* from both *latency* and *update* filters. Latency filters generate a current, expected round trip time to a remote node and update filters prevent system-level coordinate updates from spuriously affecting application behavior. Neighbor decay, in contrast, handles the problem of skewed updates that can occur when network coordinates are maintained as a passive subsystem. It allows the smooth incorporation of information from a wider range of neighbors, particularly in a system where contact between nodes is highly transient. In simulation, we confirmed that neighbor decay substantially increased stability and moderately improved continuous relative error.

5 Measuring Coordinate Systems

In this section, we review metrics used to evaluate coordinate systems and other latency services.

Relative Error. Relative error, the most basic and intuitive measure of accuracy, is the difference between the expected and actual latencies between two nodes:

$$e = \frac{|\ \|\overrightarrow{x_i} - \overrightarrow{x_j}\| - l_{ij}\ |}{l_{ij}}$$

Relative error comes in three forms: global, continuous, and neighbor. *Global relative error* is the accuracy from the viewpoint of an omniscient external viewer: at one instant, the metric is computed for all links. With the simulations that use a latency matrix, this is what we compute because we do indeed have this viewpoint. *Continuous error* is what a node computes on-the-fly as it receives new observations from remote notes. This error is added to a statistic, such as an EWMA, as in Vivaldi's confidence. Two disadvantages to continuous error are (a) a single measurement may result in a large change in value and (b) it can become skewed by a handful of remote nodes if the "working set" of active gossip is small. Instead of continuous error, we use *neighbor error* as a proxy for global error when live nodes are performing the computation themselves, *e.g.,* within live Azureus clients. Neighbor error is the distribution of relative errors for a set of recently contacted nodes. With a large number of neighbors, neighbor error generally provides a close approximation to global.

Stability. Stable coordinates are particularly important when a coordinate change triggers application activity. In our distributed streaming query system, for example, a coordinate change could initiate a cascade of events, culminating in one or more heavyweight process migrations [22]. If the systems' coordinates have not changed significantly, there is no reason to begin this process. A stable coordinate system is one in which coordinates are not changing over time, assuming that the network itself is unchanging. We use the rate of coordinate change

$$s = \frac{\sum \Delta\overrightarrow{x_i}}{t}$$

to quantify stability. The units for stability are $ms/sec.$

Descriptions of and results from other metrics are included in technical report version of this paper [15].

6 Internet-Scale Network Coordinates

Using a latency matrix can only tell part of the story of an Internet coordinate system. It helps describe the network's characteristics, *e.g.,* its intrinsic dimensionality, but misses out on problems that may occur only in a running system, such as churn, changes in latencies over time, and measurement anomalies. We used three distinct methods to understand the online performance of Azureus' coordinates: (a) on PlanetLab, we ran instrumented Azureus clients that recorded the entirety of their coordinate-related behavior (§ 6.2), (b) we crawled approximately ten thousand Azureus clients that internally tracked the performance of their coordinates using statistics we inserted into the Azureus code (§ 6.3), and (c) we

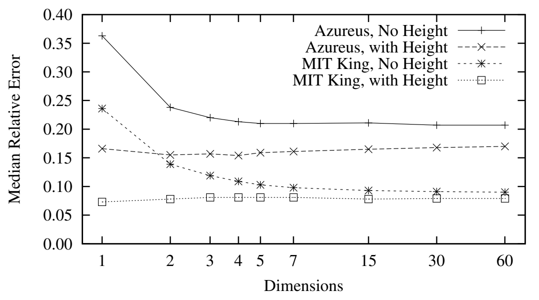


Figure 6: Because *height* had a major, positive impact on Azureus in simulation, we returned it to the *4d+h* version.

ran a benchmark to determine the effectiveness of the coordinates on application-level decisions (§ 6.4).

6.1 Refining Azureus' Coordinates

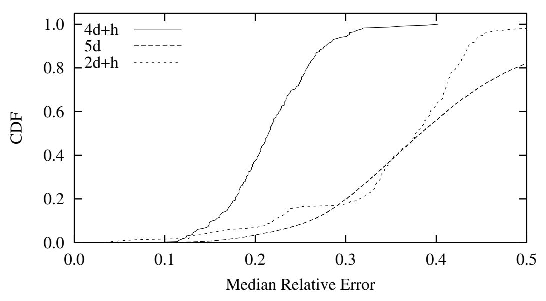
Because updates to the CVS tree could take weeks to proliferate to a majority of users, changing single variables or techniques was not feasible. Instead we relied on simulations and on-going measurement to guide the roll-out of two major coordinate versions.

Azureus' coordinates originally used two dimensions, *height*, and none of the three filtering techniques we described in Section 4. We call this version 2D+H. To create version 5D, we incorporated the two techniques from our previous research, latency and update filters, into the code. Based on our on-going PlanetLab coordinate service, which did not use height and reliably exhibited low error, we also dropped *height* and added three more dimensions. Unfortunately, removing height proved to be a mistake. Through simulations of the Azureus latency matrix (see Figure 6), we realized we could expect a substantial improvement in accuracy by converting the last dimension of the $5d$ implementation to height without changing the gossip packet structure. We also found the highly skewed neighbor sets slowed convergence and developed the *neighbor decay* technique to compensate. We combined these changes and rolled out version 4D+H.

6.2 PlanetLab Snapshots

We took snapshots of each version by running clients on approximately 220 PlanetLab nodes. Each snapshot lasted for at least three days, and logged updates with approximately $10,000$ Azureus nodes. We collected a snapshot for each of the three versions in March, July, and September 2006, respectively. Note that these instrumented clients never stored or transferred any content that travels over the Azureus network.

We compare data gathered from the different versions in Figure 7. Because the data are aggregated across roughly the same source PlanetLab nodes, the three snapshots provide a reasonable, though imperfect, way to isolate the effects of the different techniques. In all cases, we find 4D+H is more accurate and stable than both the original 2D+H and our initial rollout of 5D.



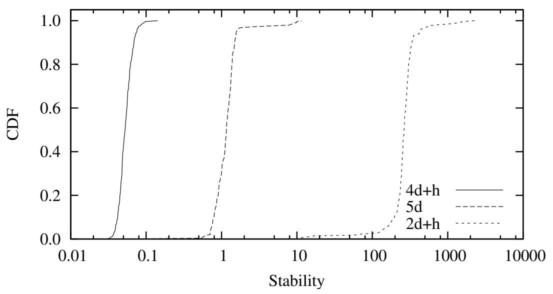


Figure 7: The combination of filtering, neighbor decay, and height lead to substantially more accurate coordinates on PlanetLab nodes participating in the Azureus network coordinate system. Comparing 2D+H to 4D+H, the data show a 43% improvement in relative error and a four orders-of-magnitude improvement in stability.

Our first revision had mixed results. Based on this data and on simulations with and without height, the data convey that the removal of height damaged accuracy more than the filters aided it. In retrospect, given the Azureus round trip time distribution (see § 3.2), in which 7.6% of the node pairs exhibit round trip times ≥ 1 second, it is not surprising that using height helped many nodes find a low error coordinate. In addition, given that two dimensions are enough to capture much of Azureus' inherent dimensionality, it is also not surprising that the addition of three dimensions did not radically improve accuracy. Although the 5D coordinates are less accurate, they are more than $2\frac{1}{2}$ orders-of-magnitude more stable because the latency filters prevent anomalous measurements from reaching the update algorithm.

Our second change was more successful. The introduction of *neighbor decay* and the re-introduction of height in 4D+H create a much more accurate coordinate space than either of the previous two snapshots. This increase in accuracy occurs because neighbor decay enables nodes to triangulate their coordinates with a larger fraction of the network (and their neighbors are doing the same) and because *height* supplies the numerous nodes on DSL and cable lines with the additional abstract distance over which all their physical communication must travel.

We first evaluated *neighbor decay* in simulation. To confirm its continued effectiveness in a live system, we

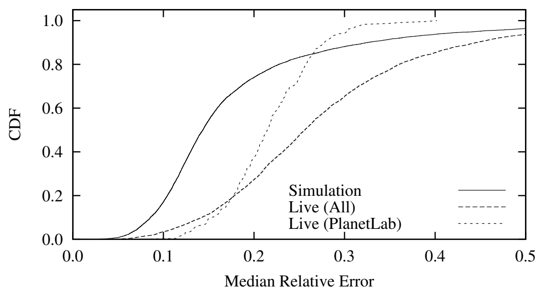


Figure 8: Reality does not live up to expectations: a comparison of probed statistics from live Azureus nodes to those from simulation suggests that accuracy could be improved by as much as 45%. Section 7 explores the major remaining impediments.

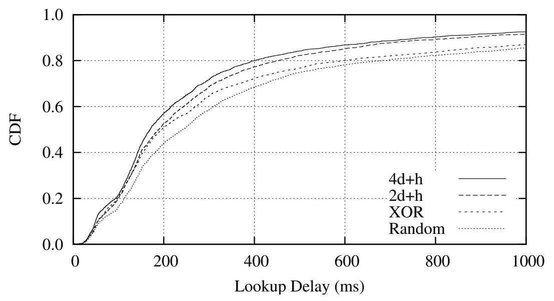


Figure 9: By choosing paths that are small detours in the logical space but lower latency, network coordinates improve lookup delay in Azureus's DHT.

performed an experiment where we monitored the convergence of a node with and without *neighbor decay* enabled as part of the 4D+H coordinate system. In an average of three trials, we found neighbor decay improved median accuracy by 35%, 40% and 54% at the 15, 30, and 60 minute marks respectively.

6.3 End-Host Live Coordinates

The logs from our Azureus clients running on PlanetLab nodes provide a detailed view of a narrow slice of the system. To obtain a picture of the broader system, we inserted online statistics collection into the Azureus CVS tree. Using its recent neighbor set, each node computed its neighbor error and stability statistics on demand when probed. We present results from Azureus end-hosts running version 4D+H.

Figure 8 "live (all)" illustrates the data from a crawl of 9477 end-hosts. We exclude live nodes with fewer than 10% of the maximum 512 neighbors because their metrics are skewed to a very small percentage of the network. The data show that the bulk of the Azureus system experiences accuracy similar to clients running on PlanetLab. However, the error on the greater Azureus network has a long tail: at the 95^{th} percentile, its accuracy is 76% worse. As we discuss in Section 7.1, we conjecture that the high rate of churn causes much of this difference in the tail.

In order to hint at the exigencies caused by running "in the wild" as opposed to safely in the lab, we compared the statistics from live Azureus nodes to those from our simulated embeddings of the Azureus latency matrix. In Figure 8, we compare live and simulated relative error. The data show a significant gap between live and simulated performance. (Prior work using the same simulator found simulations of PlanetLab mirrored live results [16].) The medians of the relative error distributions are 26% and 14% for live and simulated coordinates, respectively, a difference of 45%.

The data suggest that network coordinates have been partially tamed, but can be made substantially more accurate, and, therefore, more useful for distributed applications that would like to make cheap, quick decisions between providers of the same service. We show how the current level of accuracy affects these anycast decisions in the following section.

6.4 Application-Level Performance

Accuracy and stability metrics capture application-independent, low-level behavior. To understand how Internet-scale coordinate systems can affect application-level behavior, we also examined how Azureus uses them to make higher-level, anycast decisions in one of its common tasks: DHT key lookup. Azureus performs this operation for each tracker announcement, torrent rating lookup and publish, and NAT traversal rendezvous lookup and publish (for tunnelling through NATs).

We modified an Azureus client so that it used network coordinates to optimize lookup delay. Our experiment to evaluate the change in lookup delay first stored a set of keys in the DHT, then looked up each key using four distinct node selection methods, recording the time for the lookup operation. For each key, we ran the methods in random order.

Each method selects one node from a small set, *i.e.,* is performing an anycast: all choices will make logical progress toward the target, some have lower latency than others. Azureus uses Kademlia, which defines the logical distance between two DHT keys as the exclusive-or of their bits [17]. Starting with the logically nearest known nodes to the target: XOR picks the logically nearest node, 2D+H picks the node whose latency as predicted by the 2D+H coordinates is smallest, 4D+H picks the lowest latency node as predicted by the 4D+H coordinates, and RANDOM picks randomly from the set. Each node contacted returns its neighbors that are logically close to the target. This repeats until either a node storing the key is found or the lookup fails. Because Azureus performs DHT lookups iteratively, we were able to experiment with the lookup algorithm through code updates on only a single node.

We plot the distribution of delays from storing 250 keys and performing 2500 lookups in Figure 9. Compared to

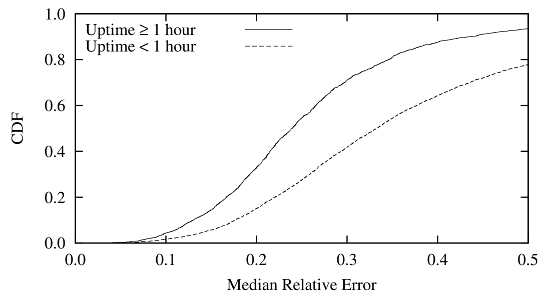


Figure 10: Azureus nodes that have been in the system for longer periods have more accurate coordinates. This suggests that churn may hurt convergence of Internet-scale coordinate systems.

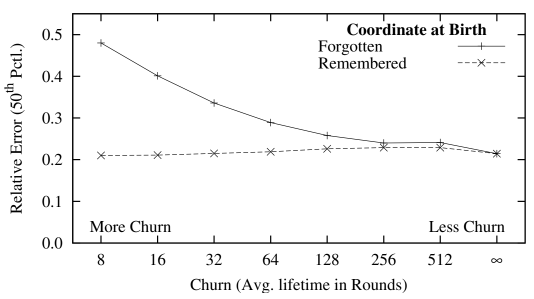


Figure 11: Coordinate systems that experience high churn rates and do not allow nodes to "remember" their previous coordinates have trouble converging.

the XOR method, which always chooses the nearest logical node, the data show that 4D+H reduces lookup delay by 33% at the 80^{th} percentile. It is 12% faster than the early version of the coordinates, 2D+H, also at the 80^{th} percentile. Because no latency prediction information is currently returned to the caller, the optimization only affects the selection of the first hop. In addition, we were not able to predict latencies to 34% of nodes due to version incompatibilities. Both of these factors suggest these improvements are conservative. We excluded lookups that timed out due to dropped UDP messages to avoid dependence on a particular timeout handling mechanism. These data show that using network coordinates can provide a substantial improvement to an application-level process.

7 Barriers to Accuracy

In this section, we examine five primary causes of the remaining difference between the current live accuracy and what appears to be achievable based on simulation results. The five barriers are: churn, drift, intrinsic error, corruption, and latency variance. We present techniques that address the first three barriers and non-malicious corruption. However, malicious corruption and latency variance remain unsolved; indeed, the latter requires a fundamentally new approach to latency prediction. Based on our simulation and PlanetLab results and on monitoring Azureus over time, we have added the techniques that address churn, drift, and non-malicious corruption to the Azureus code. While preliminary experiments suggest they function as expected, we have not yet fully quantified their effects and do not include results for them here.

7.1 Churn

Distributed network coordinate algorithms traditionally consider churn as part of their network model. Researchers ask the question: given an existing, stable system, how quickly can a new node find a stable, accurate coordinate? Unfortunately, implicit in this question is the assumption that the existing system has converged, and this assumption breaks down in many large-scale distributed systems, including Azureus. We found Azureus follows a long-tailed lifetime distribution typical of peer-to-peer systems; in its case 78% of the nodes were in the system for less than one hour.

Because coordinate updates were on the order of tens of seconds or sometimes minutes apart, nodes often did not have much time to settle into a stable position before they exited the system. Using the data from our crawl of the live network, we separated nodes into ones that had been in the system for an hour or more and those that had not. We plot the relative error experienced by these two groups in Figure 10. The data confirm that these short-lived nodes, which make up the majority of the system, are substantially less accurate than long-lived ones.

We considered three potential solutions to the problem of sustaining a coordinate system under high churn rates. First, nodes could perform a rapid initial triangulation process before shifting to a lower update rate. However, adjusting the gossip rate over time has two problems: (a) "passive" (*i.e.* maintenance-free) coordinate systems have no control over gossip and (b) in an "active" system, it would be a new, complex knob. Second, we considered "greedy optimization," where instead of just stepping once through the update process, nodes would repeat until a (local) minimum had been reached with respect to the currently known neighbors. Unfortunately, we found that this form of optimization does not work well until many neighbors are known, which is not the case early in a node's lifetime. Finally, we found a solution that is both extremely simple and had positive results in simulation: instead of starting from scratch when restarting a client, have it begin where it left off. We performed an experiment where we varied the amount of churn in simulation and toggled whether or not nodes "remembered" their coordinate on re-entry. In Figure 11, we show the results of this experiment. We found that when nodes started at the origin on re-entry, they had a deleterious effect not only on themselves, but on overall system convergence. In contrast, with this simple technique, accuracy remained about the same as when there was no churn. While this technique assumes limited drift (see next section), it appears to be a

Gravity's ρ	Migration	Error
2^6	$8ms$	25%
2^8	$17ms$	10%
2^{10}	$74ms$	10%
2^{12}	$163ms$	10%
None	$179ms$	10%

Table 1: Small amounts of *gravity* limit drift without preventing coordinates from migrating to low-error positions.

promising start to resolving the noxious effect of churn on live coordinate systems.

7.2 Drift

Monitoring our PlanetLab-based coordinate service over several months revealed that coordinates migrated in a fairly constant direction: the centroid of the coordinates did not move in a "random walk," but instead drifted constantly and repeatedly in a vector away from the origin. This was surprising because our previous study, based on a shorter, three-day trace, had not exhibited this pattern [16].

While coordinates are meant to provide *relative* distance information, *absolute* coordinates matter too. One problem with drift is that applications that use them often need to make assumptions on maximum distances away from the "true" origin. For example, one could use Hilbert functions to map coordinates into a single dimension [4]. This requires an *a priori* estimate of the maximum volume the coordinates may fill up. Mapping functions like Hilbert require that the current centroid not drift from the origin without bound. Drift also limits the amount of time that cached coordinates remain useful [12].

A "strawman" solution to drift would be to continuously redefine the origin as the centroid of the systems coordinate. Unfortunately, this would require accurate statistical sampling of the coordinate distribution and a reliable mechanism to advertise the current centroid. Our solution to drift is to apply a polynomially-increasing *gravity* to coordinates as they become farther away from the true origin. Gravity $\overrightarrow{G}$ is a force vector applied to the node's coordinate $\overrightarrow{x_i}$ after each update:

$$\overrightarrow{G} = \left(\frac{\|\overrightarrow{x_i}\|}{\rho}\right)^2 \times u(\overrightarrow{x_i})$$

where ρ tunes $\overrightarrow{G}$ so that its pull is a small fraction of the expected diameter of the network. Hyperbolic coordinates could use a similar equation to compute gravity.

Drift does not occur in simulation if one is using a latency matrix and updating nodes randomly, because this form of simulation does not capture time-dependent RTT variability. Instead, we used a 24-hour trace of our PlanetLab service to simulate the effect of gravity; we show the effect of different strengths of gravity in Table 1. The data

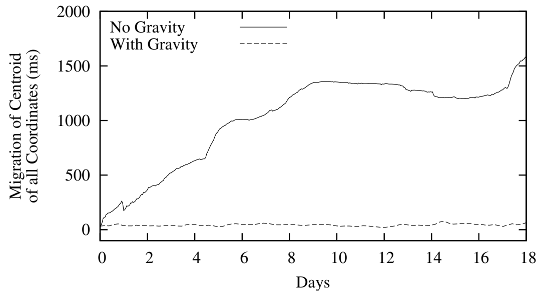


Figure 12: With *gravity*, coordinates did not drift away from their original origin as they had done before.

show that this simple technique does keep the coordinate centroid highly stationary without affecting accuracy.

To confirm the effect of gravity on a live system, we added it to our on-going PlanetLab service, which had ≈ 300 participants. In Figure 12, we compare drift before and after adding gravity over two 18 day periods. The data show that gravity effectively eliminates drift. In addition, it did not reduce accuracy, which, in both cases, had a median of about 10%. While gravity does not actively limit rotation, we did not observe a rate greater than one full rotation per three days. Determining the cause of drift is beyond the scope of this work.

7.3 Intrinsic Error

Violations of the triangle inequality occur more frequently and to a greater extent on Azureus than either on PlanetLab or for sets of DNS servers (see § 3.3). We found, perhaps surprisingly, that removing a small number of the worst violators causes a large improvement in global accuracy. Not only do the violations these nodes take part in damage their own coordinates, but the damage they cause continues to reverberate throughout the system.

We performed an experiment where we removed a small percentage of the nodes with the largest triangle violations from the Azureus latency matrix and compared this to removing a random subset of nodes of the same size. We then computed a system of coordinates and found the relative error of each link. As Figure 13 illustrates, removing only the worst 0.5 percent of nodes leads to a 20 percent improvement in global accuracy. This data parallels results from theoretical work that showed how to decrease embedding distortion by sacrificing a small fraction of distances to be arbitrarily distorted [2]. These results show that *if* a mechanism could prevent these nodes from affecting the rest of the system, it would improve overall accuracy. Two example mechanisms for node self-detection and removal from the coordinate system are: (a) directly evolving an estimate of the extent of their violations by asking neighbors for latencies to other neighbors, and (b) determining if they are subject to traffic shaping (based on the modality of their latency distribution), and therefore a major cause of triangle violations. Preliminary experi-

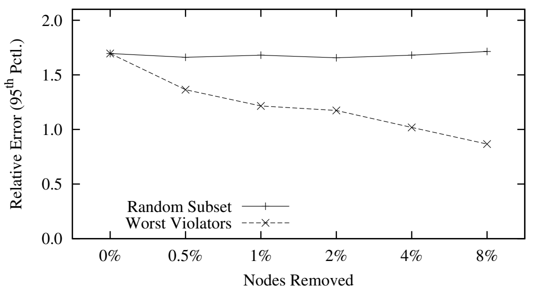


Figure 13: Removing only a small percentage of nodes with the worst triangle violations has a large effect on global accuracy.

ments with self-exclusion based on bimodality tests show an improvement in accuracy of 8% at the 95^{th} percentile.

7.4 Corruption and Versioning

An insipid fact of running a large system where users can choose when to upgrade is that not everyone is running the same version. One of the problems we found with our original deployments was that about 13% of the remote coordinates received during gossip were at the origin; that is, $[0]^d$. After much discussion (Is that incredible churn rate possible? Do nodes behind firewalls never update their coordinates?), we realized that this problem was due to a portion of the network running an old code version. In fact, during one crawl of the Azureus network, we found only about 44% of the $\approx$ 9000 clients crawled were using the current version. While not very exciting, realizing this fact allowed us to compensate for it both in the coordinate update process and in active statistics collection through the explicit handling of different versions within the code.

Kaafar *et al.* have begun investigating the more interesting side of the problem of coordinate corruption: malicious behavior [14]. They divide attacks into four classes: disorder, isolation, free-riding, and landmark control. While we did not see any evidence of intentionally corrupt messages, it would be trivial to install a client, or a set of clients, that responded with random values, for example (just as the MPAA runs clients with spurious content advertisements to squelch piracy). As Internet-scale coordinate systems come into wider use, they will need to grapple with both oblivious and malicious corruption.

7.5 Latency Variance

The prior "barriers to accuracy" paint a rosy picture; most problems have a fairly simple solution that practitioners can use to build more accurate, live coordinate systems. The existence of wide variation in latency measurements between the same pair of nodes over a short period of time is a harder problem with broad ramifications. If variances are very large what does it actually mean to "predict" the latency from one node to another? Using the data from our longest snapshot (5D), we determined the standard deviation of latency between each pair of nodes. We found that round trip times varied by a *median* of $183ms$. This spread affects other latency prediction systems as well. A reactive measurement service, such as Meridian, will be more error-prone or have higher overhead if small numbers of pings do not sufficiently measure the latency to a high variance target. In fact, coordinate systems may be in a better position to address this problem because they can retain histories of inter-node behavior.

As reviewed in Section 4.1, we developed *latency filters* in previous work. They act as a low-pass filter: anomalies are ignored while a baseline signal passes through. Additionally, they adapt to shifts in the baseline that BGP route changes cause, for example. These filters assign a link a *single* value that conveys the expected latency of the link. While we found these simple filters worked well on PlanetLab, describing a link with a single value is not appropriate with the enormous variance we observe on some of Azureus' links.

We ran an experiment where we compared ICMP, filtered, and raw latency measurements that were taken at the same time. To determine which destination nodes to use, we started Azureus on three PlanetLab nodes and chose five ping-able neighbors after a twenty-minute startup period. We then let Azureus continue to run normally for six hours while simultaneously measuring the latency to these nodes with *ping*. We plot the data in Figure 14. Figure 14 (a) illustrates a pair similar to our PlanetLab observations: there was raw application-level and ICMP variance, but a consistent baseline that could be described with a single value. In contrast, Figure 14 (b) portrays a high variance pair: while the filter does approximate the median round trip time, it is difficult to say, at any point in time, what the latency is between this pair.

The impact of the dual problems of high latency variance and modifying algorithms to deal with high latency variance is not limited to network coordinate systems. Latency and anycast services deployed "in the wild" need to address this problem. While there may exist methods to incorporate this variance into coordinate systems — either through "uncertainty" in the latency filters or in the coordinates themselves — resolving this problem is beyond the scope of this paper.

8 Related Work

Early work on latency prediction services focused on reducing the intractability of all-pairs measurements through clustering. Based on the assumption that nodes in the same cluster would have similar latencies to nodes in another cluster, researchers examined how to create accurate clusters and how to minimize inter- and intra-cluster measurement overhead. Francis *et al.* created clusters based on IP address prefixes, but found that prediction

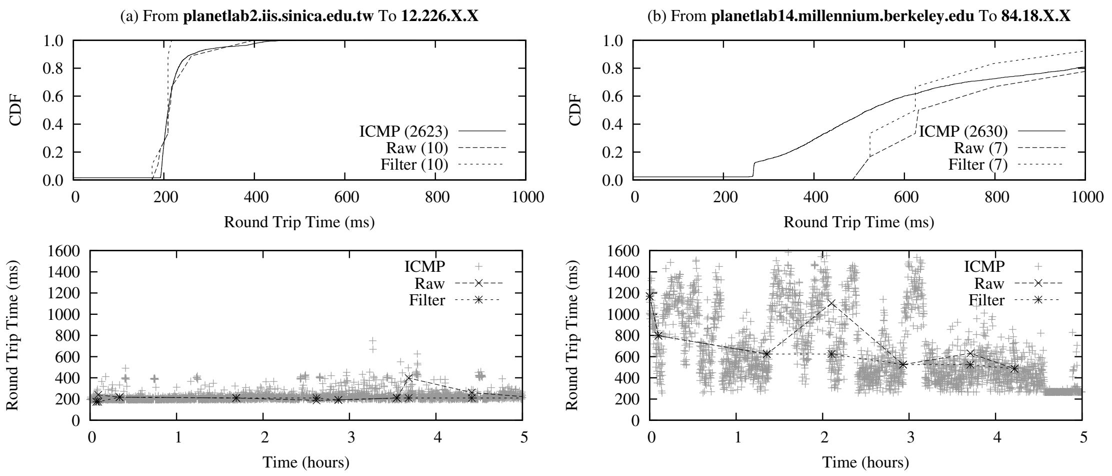


Figure 14: A comparison of round trip times between two sets of node pairs using ICMP, raw application-level measurements, and filtered measurements. Pair (a) exhibits some variance, but shows a consistent baseline. With pair (b), the variance is so large that assigning this node a coordinate — or putting it into a consistent Meridian ring — is bound to be an error-prone process. The number in parentheses in the legend is the number of round trip time measurements in the cumulative distribution function.

error was heavily dependent on the initial choice of representatives [11]. Chen *et al.* addressed this problem through the automatic formation of clusters and representatives; they found the cluster size and, more generally, the amenability of the network to clustering had a large effect on accuracy [5]. Ratnasamy *et al.* proposed a hybrid approach: nodes that are similar distances away from fixed landmarks place themselves into the same cluster; they also found error was highly dependent on the number of bins [23]. Because all of this clustering involves measurement and lower network layers are already performing much of this measurement, Nakao *et al.* proposed reducing overhead by tapping into this existing information; unfortunately, this requires a change in the interface of Internet routers [18].

While this research and the work on network coordinates that grew out of it focus on generalized latency prediction — maintaining an infrastructure that works well for most queries — a separate body of work has focused more directly on the problem of finding the nearest of many replicated services. In direct response to an application-level request, Meridian finds the nearest overlay node (*i.e.,* one running Meridian) to an arbitrary point in the Internet through a set of pings that progress logarithmically closer to the target [29]. Freedman *et al.* developed OASIS, a distributed service explicitly designed to help clients find and choose a "good" server out of many [12]. Building on Meridian, OASIS primarily focuses on network locality, but also incorporates liveness and load. OASIS employs a reliable core of hosts to map clients to nearby servers, which are assumed to be long-lived. Note the distinct purposes of these anycast services from those of network coordinates: Meridian and OASIS are designed for the case where contact with the service will be frequent and long-lived enough to outweigh the high upfront cost of finding the best service. With their current levels of accuracy (good but not perfect) and maintenance (zero), network coordinates fall to the other side of the trade-off: short-lived, cheap decisions for which finding the exact answer is not worthwhile, but repeatedly finding a good answer leads to aggregate savings. While Meridian (and OASIS) are inherently reactive – acting in response to a query – they too could be more tightly integrated with an application, using its messages to dampen ring maintenance, for example.

8.1 Network Coordinates

There exist two main classes of algorithms for calculating coordinates: *landmark-based* schemes, in which overlay nodes use a fixed number of landmark nodes to calculate their coordinates, and *simulation-based* schemes, which are decentralized and calculate coordinates by modeling nodes as entities in a physical system.

Landmark-based. In *GNP* [19], nodes contact multiple landmark nodes to triangulate their coordinates. The drawbacks of this approach are that the accuracy of the coordinates depends on the choice of landmark nodes and landmark nodes may become a bottleneck. *Lighthouses* [21] addresses this by supporting multiple independent sets of landmarks with their own coordinate systems. These local coordinates map into a global coordinate system. *PIC* [7] does not use explicit landmarks, incorporating measurements to any node using a simplex optimization algorithm to obtain an up-to-date coordinate. These landmark-based schemes require a reasonably stable infrastructure and, to the best of our knowledge, have not

Octant: A Comprehensive Framework for the Geolocalization of Internet Hosts

Bernard Wong, Ivan Stoyanov, Emin Gün Sirer
Dept. of Computer Science, Cornell University, Ithaca, NY 14853

ABSTRACT

Determining the physical location of Internet hosts is a critical enabler for many new location-aware services. In this paper, we present Octant, a novel, comprehensive framework for determining the location of Internet hosts in the real world based solely on network measurements. The key insight behind this framework is to pose the geolocalization problem formally as one of error-minimizing constraint satisfaction, to create a system of constraints by deriving them aggressively from network measurements, and to solve the system geometrically to yield the estimated region in which the target resides. This approach gains its accuracy and precision by taking advantage of both positive and negative constraints, that is, constraints on where the node can and cannot be, respectively. The constraints are represented using regions bounded by Bézier curves, allowing precise constraint representation and low-cost geometric operations. The framework can reason in the presence of uncertainty, enabling it to gracefully cope with aggressively derived constraints that may contain errors. An evaluation of Octant using PlanetLab nodes and public traceroute servers shows that Octant can localize the median node to within 22 mi., a factor of three better than other evaluated approaches.

1 INTRODUCTION

Determining the physical location of an Internet host is a key enabler for a wide range of network services. For example, mapping nodes to locations on the globe enables customized content on the web, simplifies network management in large installations, and aids network diagnosis. Accurately determining the position of a node in the real world based solely on network measurements, however, poses many challenges. The key obstacles to accurate and precise geolocalization are threefold: how to represent network locations for nodes, how to extract constraints on where nodes may or may not be located, and how to combine these constraints to yield good estimates of node position [1].

In this paper, we present a novel, comprehensive framework for geolocating Internet hosts called Octant [2]. Octant provides an intuitive and generic framework which represents node positions precisely using regions, expresses constraints succinctly as areas, and computes positions accurately by solving a system of geometric constraints. A small number of landmarks whose positions are approximately known anchors the constraint system to the physical globe. The Octant approach is comprehensive and general; it enables almost all past work on geolocalization to be expressed within the framework, as a (limited) subset of the techniques described in this paper.

Past approaches to geolocalization on the Internet rely solely on *positive information*, that is, information on where a node may be located. For instance, a landmark that pings the target may conclude that the target must lie within a disk centered around the landmark whose radius is bounded by the speed of light times the one-way ping latency. In addition to such positive information, Octant can take advantage of *negative information*, that is, information on where the node may not be located. For instance, momentarily assuming an ideal network with no queuing delays, Octant enables a landmark that measures a high ping latency to express the fact that the node is at least a minimum distance away from the landmark.

Octant represents the potential area where a node may be located explicitly as a surface bounded by Bézier curves. In contrast with past work that keeps track of and computes a single position estimate, Octant's geolocalization yields a set of points expressed as an enclosed, potentially non-convex and disconnected area where the node might lie. The Bézier curve representation enables these areas to be expressed precisely in a compact manner, and boolean operations on areas such as union, intersection, and subtraction are computed efficiently. We outline a Monte Carlo technique for selecting a single, representative point estimate from such sets to facilitate comparisons with past work and to support legacy applications which expect a location estimate consisting of a single point. In practice, Octant's location estimates are accurate, that is, they almost always contain the actual physical location of the target in the estimated area, as well as precise, that is, the size of the estimated area is small.

Since networks rarely follow idealized models of transmission, the fidelity of geolocation schemes are fundamentally limited by how aggressively they mine the network for constraints. Given the relatively high variance in the correlation between network latency and geographical distance due to congestion and indirect routing, extracting useful positive and negative information is a challenge. Octant uses various principled methods to extract precise constraints from noisy Internet measurements. It compensates for dilation stemming from inelastic delays incurred on the last hop by computing an extra "height" dimension, that captures the effects. It minimizes the impact of indirect routes through piecewise localization of routers on the network path, where it localizes ordinary routers on the path and uses their approximate location to further refine the position estimate of the target node. Finally, Octant uses a weighted solution technique where weights correspond to confidence in a derived constraint to enable the use of aggressive constraints in addition to conservative ones without creating a non-solvable constraint system.

The Octant framework is general and comprehensive. Where available, data from the WHOIS database, the DNS names of routers, and the known locations of uninhabited regions can be naturally integrated into the solution to refine it further. Interestingly, this optimization has enabled Octant to identify "misnamed" routers whose DNS names, based on their ISP's naming convention, would indicate that they are in a particular state when in reality they are several hundred miles away (and are named after a node with which they peer).

Overall, this paper presents a geolocalization system for determining the physical location of hosts on the Internet, and makes three contributions. First, this paper provides a novel and general framework for expressing location constraints and solving them geometrically to yield location estimates. The solution technique, based on Bézier-regions, provides a general-purpose foundation that accommodates any geographic constraint. Second, the paper shows how to aggressively extract constraints from network latency data to yield highly accurate and precise location estimates. Finally, the paper describes a full implementation of the Octant framework, evaluates it using measurements from PlanetLab hosts as well as public traceroute servers, and compares directly to past approaches to geolocalization. We show that the system achieves a median accuracy of 22 miles for its position estimates. In contrast, the best accuracy achieved by GeoLim [11], GeoPing [15], and GeoTrack [15] achieves a median accuracy of 70 miles. Overall, the system is practical, provides a location estimate in under a few seconds per target and achieves high accuracy and precision.

2 RELATED WORK

Past work on geolocalization can be broken down into approaches that determine a single point estimate for a target, and those that, like Octant, provides a region encompassing the set of points where the target may lie.

2.1 SINGLE-POINT LOCALIZATION

IP2Geo [15] proposes three different techniques for geolocalization, called GeoPing, GeoTrack and GeoCluster. GeoPing maps the target node to the landmark node that exhibits the closest latency characteristics, based on a metric for similarity of network signatures [6]. The granularity of GeoPing's geolocalization depends on the number and location of the landmarks, requiring a landmark to be close to each target to produce low-error geolocation.

GeoTrack performs a traceroute to a given target, extracts geographical information from the DNS names of routers on the path, and localizes the node to the last router on the path whose position is known. The accuracy of GeoTrack is thus highly dependent on the distance between last recognizable router to the landmark, as well as the accuracy of the positions extracted from router names.

GeoCluster is a database based technique that first breaks the IP address space into clusters that are likely to be geographically co-located, and then assigns a geographical location to each cluster based on IP-to-location mappings from third party databases. These databases include the user registration records from a large web-based e-mail service, a business web-hosting company, as well as the zip-codes of users of an online TV program guide. This technique requires a large, fine-grain and fresh database. Such databases are not readily available to the public due to potential privacy concerns, the clustering may not sufficiently capture locality, the accuracy of such databases must be perpetually refreshed, and, most importantly, the overall scheme is at the mercy of the geographic clustering performed by ISPs when assigning IP address ranges.

Services such as NetGeo [14] and IP2LL [1] geolocalize an IP address using the locations recorded in the WHOIS database for the corresponding IP address block. The granularity of such a scheme is very coarse for large IP address blocks that may contain geographically diverse nodes. The information in the WHOIS database is also not closely regulated and the address information often indicates the location of the head office of the owner which need not be geographically close to the actual target. Quova [3] is a commercial service that provides IP geolocalization based on its own proprietary technique. Neither the details of the technique nor a sample dataset are publicly available.

There are several graphical traceroute tools that offer the geographical location of each intermediate router. GTrace [16] successively uses DNS LOC entries, a proprietary database of domain name to geographical location mappings, NetGeo, and domain name country codes, as available, to localize a given node. VisualRoute [4] is a commercial traceroute tools that also offer geographic localization of the nodes along the path.

2.2 REGION LOCALIZATION

GeoLim [11] derives the estimated position of a node by measuring the network latency to the target from a set of landmarks, extracts upper bounds on position based on inter-landmark distance to latency ratios, and locates the node in the region formed by the intersection of these fixes to established landmarks. Since it does not use negative information, permit non-convex regions or handle uncertainty, this approach breaks down as inter-landmark distances increase.

In contrast, Octant provides a general framework for combining both positive and negative constraints to yield a small, bounded region in which a node is located. It differs from past work in that it enables negative information to be used for localization, separates the selection of a representative point estimate from the computation of the feasible set of points in which a node might be located, permits non-convex solution areas, and aggressively harvests constraints from network latency measurements.

Topology-based Geolocation (TBG) [13] uses the maximum transmission speed of packets in fiber to conservatively determine the convex region where the target lies from network latencies between the landmarks and the target. It additionally uses inter-router latencies on the landmarks to target network paths to find a physical placement of the routers and target that minimizes inconsistencies with the network latencies. TBG relies on a global optimization that minimizes average position error for the routers and target. This process can introduce error in the target position in an effort to reduce errors on the location of the intermediate routers. Octant differs from TBG by providing a geometric solution technique rather than one based on global optimization. This enables Octant to perform geolocalization in near real-time, where TBG requires significantly more computational time and resources. A geometric solution technique also allows Octant to seamlessly incorporate exogenous geometric constraints stemming from, for example, geography and demographics. This provides Octant with more sources of information for its geolocalization compared to TBG.

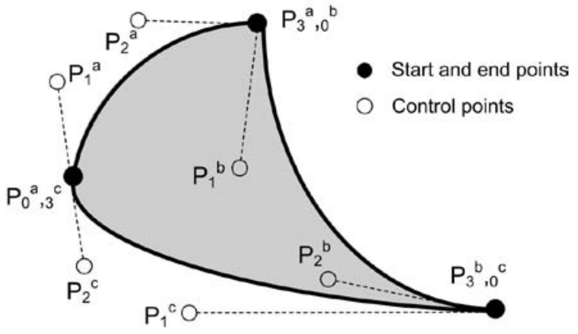


Figure 1: Location representation in Octant. Octant represents the estimated target location as a region bounded by a set of Bézier curves. Each curve a, b, c consists of four control points $P_0, ..., P_3$ with P_0 and P_3 as the start and end points respectively and P_1 and P_2 as control points that help direct the curve. This figure requires a total of only nine control points to precisely define. Bézier curves provide a compact way to represent large, complex areas precisely. They also admit efficient intersection, union, and subtraction operations.

Localization has been studied extensively in wireless systems. The wireless localization problem, however, is significantly different from, and easier than, localization on the Internet, as air is close to a perfect medium with well-understood transmission characteristics. The most comprehensive work on localization in wireless networks is Sextant [12]. We share with Sextant the basic insight for accommodating both positive and negative constraints and enabling constraints to be used by landmarks whose positions are not known definitively. Octant differs substantially from Sextant in the various mechanisms it uses to translate Internet measurements to constraints, including its mapping of latencies to constraints, isolating last hop delays, and compensating for indirect routes, among others.

3 FRAMEWORK

The goal of the Octant framework is to compute a region β_i that comprises the set of points on the surface of the globe where node i might be located. This *estimated location region* β_i is computed based on constraints $\gamma_0 \ldots \gamma_n$ provided to Octant.

A constraint γ is a region on the globe in which the target node is believed to reside, along with an associated weight that captures the strength of that belief. The constraint region can have an arbitrary boundary, as in the case of zipcode information extracted from the WHOIS database or coastline information from a geographic database. Octant represents such areas using Bézier-regions, which consist of adjoining piecewise

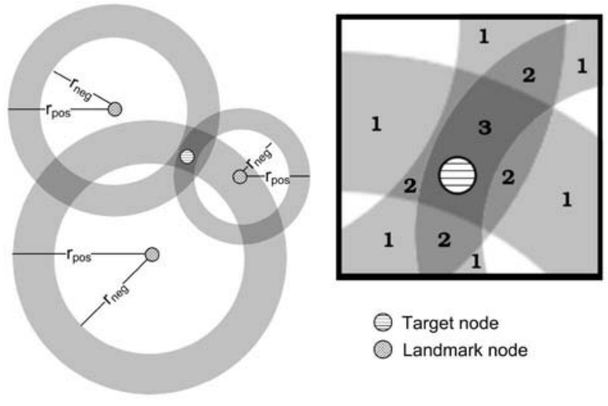


Figure 2: Octant computes an estimated location region for a target node by combining positive and negative information available through latency measurements. The resulting location estimate comprises non-convex, potentially disjoint regions separated by weight.

Bézier curves as illustrated in Figure 1. Bézier curves are polynomial parametric curves with $n + 1$ control points $P_0, ..., P_n$ where n is the order of the polynomial, with $n = 3$ for most implementations. Intuitively, the points P_0 and P_n are the start and end points with the remaining points providing the directional information. Bézier regions provide both precise and compact representation of complex shapes. For example, a circle can be represented exactly using four adjoining Bézier curves and a total of twelve control points.

Typically constraints are obtained via network measurements from a set of nodes, called *landmarks*, whose physical locations are at least partially known. Every landmark node L_j has an associated estimated location region β_{L_j}, whose size captures the amount of error in the position estimate for the landmark. We call a node a *primary landmark* if its position estimate was created via some exogenous mechanism, such as a GPS measurement or by mapping a street address to global coordinates. Typically, primary landmarks have very low error associated with their position estimates. We call a node a *secondary landmark* if its position estimate was computed by Octant itself. In such cases, β_{L_j} is the result of executing Octant with the secondary landmark L_j as the target node.

Octant enables landmarks to introduce constraints about the location of a target node based either on *positive* or *negative* information. A positive constraint is of the form "node A is within x miles of Landmark L_1," whereas a negative constraint is a statement of the form "node A is further than y miles from Landmark L_1." On a finite surface, such as the globe, these two statements both lead to a finite region in which the node is believed to lie. However, the nature of the constraint, either positive or negative, makes a big difference in how these regions are computed.

In the simple case where the location of a primary landmark is known with pinpoint accuracy, a positive constraint with distance d defines a disk with radius d centered around the landmark in which the node must reside. A negative constraint with distance d' defines the complement, namely, all points on the globe that are not within the disk with radius d'. In typical Octant operation, each landmark L_j contributes both a positive and a negative constraint. When the source landmark is a primary whose position is known accurately, such constraints define an annulus.

Octant enables meaningful extraction of constraint regions even when the position of the landmark is approximate and consists of an irregular region. For a secondary landmark k whose position estimate is β_k, a positive constraint with distance d defines a region that consists of the union of all circles of radius d at all points inside β_k (formally, $\gamma = \bigcup_{(x.y)\in\beta_k} c(x, y, d)$ where $c(x, y, d)$ is the disk with radius d centered at (x, y)). In contrast, a negative constraint rules out the possibility that the target is located at those points that are within distance d regardless of where the landmark might be within β_k (formally, $\gamma = \bigcap_{(x,y)\in\beta_k} c(x, y, d)$).

Given the description above, it may seem that computing these intersection and union regions might take time proportional to the area of β_k, and thus be infeasible. Octant's representation of regions using Bézier curves enables these operations to be performed very efficiently via transformations only on the endpoints of Bézier segments. Since Bézier curves are used heavily in computer graphics, efficient implementations of Bézier clipping and union operations are available. However, the number of Bézier segments in a region increases with each intersection and union operation, which gradually expands the number of curves to track and manipulate, which in turn poses a limit to the scalability of the framework. So a scalable Octant implementation may decide to approximate certain complex β_k with a simple bounding circle in order to keep the number of curves per region in check and thus gain scalability at the cost of modest error. Figure 3 illustrates the derivation of positive and negative constraints from primary and secondary landmarks.

Given a set Ω of positive constraints and a set Φ of negative constraints on the position of a target node i, the estimated location region for the target is given by:

$$\beta_i = \bigcap_{X_i\in\Omega} X_i \setminus \bigcup_{X_i\in\Phi} X_i.$$

This equation is precise and lends itself to an efficient and elegant geometric solution. Figure 2 illustrates how

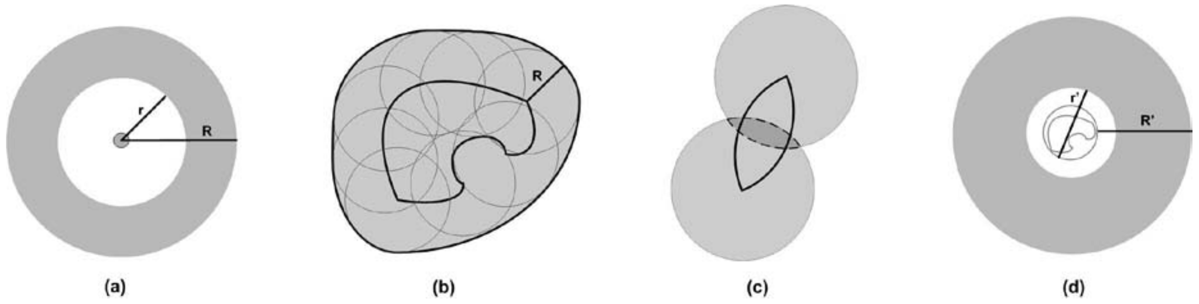


Figure 3: Comprehensive use of positive and negative constraints in Octant. (a) A primary landmark, with a precise position estimate, and its associated constraints. (b) Positive constraints are calculated by taking the union of all circles in the estimated area. A node within distance d must reside in the region marked with the dark outer line. Only a subsample of the circles are shown for clarity. (c) Negative constraints are computed by taking the intersection of all circles in the estimated area. A node outside of distance d can not be in the region marked with the dotted line. (d) A secondary landmark, whose position is not known precisely. Note that the associated constraints lead to a larger annulus, due to the conservative, sound way in which Octant combines them. An implementation may replace complex Bézier regions with a bounding circle for efficiency.

Octant combines constraints to yield an accurate estimated location region for a target.

There are, however, many issues to solve before this approach can be used for practical geolocalization on the Internet. In the general formulation above, all constraints are weighted equally and the solution is discrete; a point is either part of the solution space or it is not. A discrete solution strategy leads to a brittle system, as a single erroneous constraint will collapse the estimated location region down to the empty set. One strategy is to use only highly conservative positive constraints derived from the speed of light and avoid all negative constraints. We show later that such a conservative strategy does not achieve good precision. In the next set of sections, we detail optimizations that enable the basic Octant framework to be applied to noisy and conflicting measurements on the Internet.

If latencies on the Internet were directly proportional to distances in the real world, the geolocalization problem would be greatly simplified. Three factors complicate Internet latency measurements. First, the Internet consists of heterogeneous links, hosts and routers whose transmission and processing speeds vary widely. Second, inelastic delays incurred on the last hop can introduce additional latencies that break the correspondence between round trip timings and physical distances. Finally, packets often follow indirect, circuitous paths from a source to a destination, rendering great-circle approximations inaccurate. In the next three sections, we address each of these problems in turn.

3.1 MAPPING LATENCIES TO DISTANCES

The network latency between a target and a landmark physically bounds their maximum geographical distance. A round-trip latency measurement of d milliseconds between a landmark and a target can be translated into a distance constraint using the propagation delay of light in fiber, approximately $\frac{2}{3}$ the speed of light. This yields a conservative positive constraint on node locations that can then be solved using the Octant framework to yield a sound estimated position for the target; such an estimate will never yield an infeasible ($\emptyset$) solution. In practice, however, such constraints are so loose that they lead to very low precision.

Yet the correlation between latency measurements and real-world distances is typically better and tighter than constraints based on the speed of light. Figure 4 plots the network latency against physical distance from a primary landmark (planetlab1.cs.rochester.edu) to all other primary landmarks in our study. The figure makes clear the loose correlation between physical distance and illustrates how overly conservative the speed of light bounds can be. In addition, the empty region to the lower right suggests that few links are significantly congested; nodes that are physically close are typically reachable in a short amount of time. This presents an opportunity for a system wishing to aggressively extract constraints at the risk of occasionally making overly aggressive claims, to both tighten the bounds on positive constraints and to introduce negative constraints.

Octant calibrates each landmark when the landmark is initialized as well as periodically to determine the correlation between network measurements performed from that landmark and real-world distances. The goal of the calibration step is to compute two bounds $R_L(d)$ and $r_L(d)$ for each landmark L and latency measurement d such that a node i whose ping time is d will be between

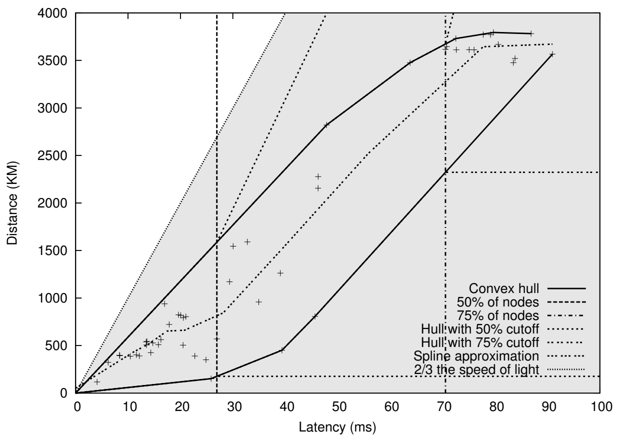


Figure 4: The latency-to-distance plot of peer landmarks for a representative landmark (planetlab1.cs.rochester.edu). The shaded region denotes the valid point locations as bounded by the propagation delay time of light in fiber. The convex hull around the data-points serves as the positive and negative constraints for the node. For a given latency, the top and bottom of the hull represent the outer and inner radius respectively of the constraint annulus. As distances increase, fewer representative nodes remain, rendering the convex hull overly aggressive. Vertical lines indicate the 50 and 75th percentile cutoffs, where the convex hull is cut and replaced with conservative positive and negative constraints when insufficient representative nodes remain.

$r_L(d) \leq ||loc(L) - loc(i)|| \leq R_L(d)$. This permits Octant to extract a positive and a negative constraint for each measurement made from each landmark. Note that when $r_L(d) = 0$, the negative constraint is defunct and does not play a role in localization; for nodes that are so close that ping times are dominated by queuing delays, r_L should be zero.

A principled approach is used to conservatively pick R_L and r_L. Each landmark periodically pings all other landmarks in the system, creating a correlation table much like Figure 4. It then determines the convex hull around the points on the graph. Functions R_L and r_L correspond to the upper and lower facets of the convex hull. This approach for extracting constraints is both tight and conservative. The R_L and r_L bounds do not contradict any empirical results, as the convex hull envelopes all data points measured at the landmark. The bounds are significantly tighter than bounds derived from linear functions used in previous techniques [11]. And the convex hull facets are smooth, positively sloped, and closely track the average latency to distance correlation.

In practice, this approach yields good results when there are sufficient landmarks that inter-landmark measurements approximate landmark-to-target measurements. In cases where the target has a direct and congestion-free path to the landmark, it may lie beyond $R_L(d)$, and vice versa for $r_L(d)$. While extensions to Octant we discuss later can compensate for occasional errors, the r and R estimates may be systematically wrong when there are just insufficient landmarks to draw statistically valid conclusions. Consequently, Octant introduces a cutoff at latency ρ, such that a tunable percentile of landmarks lie to the left of ρ, and discards the part of the convex hull that lies to the right of ρ. That is, only the part of the convex hull for which sufficient data points are available is taken into consideration. Octant then uses $r_L(x) = r_L(\rho), \forall x \geq \rho$, and $R_L(x) = m(x-\rho) + R_L(\rho), m = (y_z - R_L(\rho))/(x_z - \rho)$, where a fictitious sentinel datapoint z, placed far away, provides a smooth transition from the aggressive estimates on the convex hull towards the conservative constraints based on the limits imposed by the speed of light.

3.2 LAST HOP DELAYS

Mapping latencies to distances is further complicated by additional queuing, processing, and transmission delays associated with the last hop. For home users, these last hop delays can be attributed to cable and DSL connections that are often under-provisioned. Even in the wide area, the processing overhead on servers adds additional time to latency measurements that can overshadow the transmission delays. For instance, on overloaded Planetlab nodes, measured latencies can be significantly inflated even with careful use of kernel timestamps. Consequently, achieving more accurate and robust localization results requires that we isolate the inelastic delay components which artificially inflate latency measurements and confound the latency to distance correlation.

Ideally, a geolocalization system would query all routers on all inter-node paths, isolate routers that are present on every path from each node, and associate the queuing and transmission delays of these routers along with the node's average processing delay as the inelastic component of the node. Since this approach is impractical, we need a feasible way to approximate the last hop delay from latency measurements.

Three properties of the problem domain motivate an end-to-end approach to the measurement and representation of last hop delay in Octant. First, localization needs to be performed quickly without the cooperation of the target host. This rules out the use of precise timing hardware for packet dilation, as well as software approaches that require pre-installed processing code on the target. Second, creating detailed maps of the underlying physical network, as in network tomography [19, 8], entails significant overhead and does not yet provide answers on the timescales necessary for on-the-fly localization. Third, Octant has mechanisms in place to accommodate uncertainty in constraints (section 3.4) and can thus afford imprecision in its last hop delay estimates. These properties led us to use a fast, low-overhead, end-to-end approach for capturing the last hop delay seen on measurements from a given host in a single, simple metric which we call height.

Octant derives height estimates from pair-wise latency measurements between landmarks. Primary landmarks, say a, b, c, measure their latencies, denoted $[a, b]$, $[a, c]$, $[b, c]$. The measure for latency is the round-trip time, which captures the last hop delays in both link directions. Since the positions of primary landmarks are known, the great circle distances between the landmarks can be computed, which yield corresponding estimates of transmission delay, denoted (a, b), (a, c), (b, c). This provides an estimate of the inelastic delay component between any two landmarks [3]; for instance, the inelastic delay component between landmarks a and b is $[a, b] - (a, b)$. Octant determines how much of the delays can be attributed to each landmark, denoted a', b', c', by solving the following set of equations:

$$\begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} a' \\ b' \\ c' \end{bmatrix} = \begin{bmatrix} [a,b]-(a,b) \\ [a,c]-(a,c) \\ [b,c]-(b,c) \end{bmatrix}$$

Similarly, for a target t, Octant can compute t',as well as an estimate of the longitude and latitude, t_{long} and t_{lat}, by solving the following system of equations:

$$\begin{aligned} a' + t' + (a,t) &= [a,t] \\ b' + t' + (b,t) &= [b,t] \\ c' + t' + (c,t) &= [c,t] \end{aligned}$$

where (a, t) can be computed in terms of a_{long}, a_{lat}, t_{long}, t_{lat}. We can then solve for the t', t_{long}, t_{lat} that minimizes the residue. The computed t_{long} and t_{lat} result, similar to the synthetic coordinates assigned by [9], has relatively high error and is not used in the later stages. The target node itself need not participate in the solution for its height, except by responding to pings from landmarks. Figure 5 shows the heights of the landmarks in our PlanetLab dataset.

Given the target and landmarks' heights, each landmark can shift its R_L up if the target's height is less than the heights of the other landmarks, and similarly shift its r_L down if the target's height is greater than the heights of the other landmarks. This provides a principled basis for ensuring that at least a fraction of the time packets spend in the last hop do not skew the derived constraints.

3.3 INDIRECT ROUTES

The preceding discussion made the simplifying assumption that route lengths between landmarks and the target

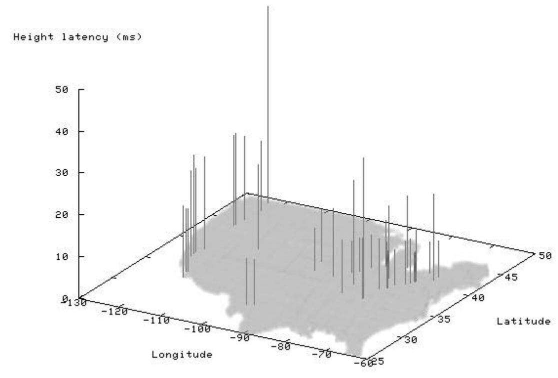


Figure 5: Heights computed by Octant to capture last hop delays on network paths to geographically distributed landmarks. Vertical bars represent landmarks, their position corresponds to their physical location, while the length of the bars corresponds to their assigned heights.

are proportional to great circle distances. Studies [17] have shown that this is often not the case in practice, due to policy routing. For instance, routes between subscribers in Ithaca, NY and Cornell University traverse Syracuse, NY, Brockport, IL, and New York City before getting routed back to Cornell, traveling approximately 800 miles to cover less than a mile of physical distance. A geolocalization system with a built-in assumption of proportionality would not be able to achieve good accuracy.

Note that the preceding section on height computation addresses some, but not all, of the inaccuracies stemming from indirect routes. In the example above, if all packets from this landmark get routed through Syracuse, NY, the distance between Ithaca and Syracuse will be folded into the landmark's height, enabling the landmark to accurately compute negative information even for nodes that are near it (without the height, the landmark might preclude its next door neighbors from being located in Ithaca). The height optimization, however, does not address inaccuracies stemming from the inconsistent, or unexpected use of indirect routes. Specifically, nodes with otherwise low heights might choose unexpectedly long and circuitous routes for certain targets. This occurs often enough in practice that accurate geolocalization requires a mechanism to compensate for its effects.

Octant addresses indirect routes by performing piecewise localization, that is localizing routers on the network path from the landmarks to the target serially, using routers localized on previous steps as secondary landmarks. This approach yields much better results than

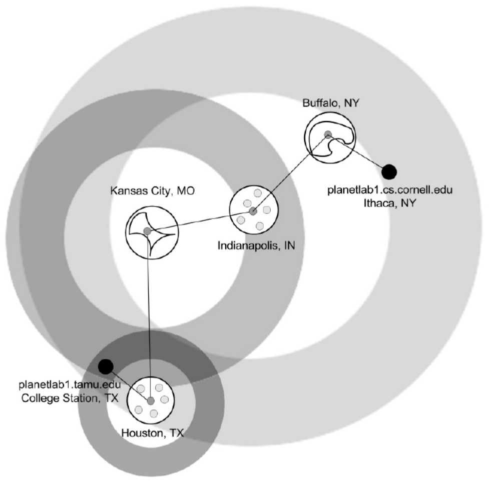


Figure 6: The use of intermediate routers as secondary landmarks can significantly increase localization accuracy. Octant is used first to determine the estimated location area for a router. Where possible, Octant refines location estimates based on the city, extracted from the router's DNS name with **undns**, in which the router is located. This is shown for Indianapolis and Houston, where dots represent the center of every zipcode located in that city. For Buffalo and Kansas City, the location of the routers is computed using Octant without undns information. The rings around Buffalo and Ithaca are omitted for clarity.

using just end-to-end latencies, and is illustrated in Figure 6. Since Octant can perform localization based solely on round-trip timings, localizing routers does not require any additional code to be deployed.

While the Octant framework can be used as is to locate routers, the structured way in which most routers are named enables Octant to extract more precise information about their location. Octant performs a reverse DNS lookup on each router on the path and determines the city in which it resides by using the **undns** [18] tool, which extracts locations from ISP-specific naming patterns in router names. The city names for routers with city information are converted into geographical coordinates using data from the US census zipcode database. A given city can have multiple coordinates in the database, with each representing the location of a zipcode region. The location of a router of a given city is the bounding circle encompassing the city's coordinates with a tunable slack to account for large zipcode regions. This approach yields much better results than using just end-to-end latencies, as routes between routers separated by a single link is largely void of indirect routing.

3.4 HANDLING UNCERTAINTY

A mechanism to handle and filter out erroneous constraints is critical to maintaining high localization accuracy. The core mechanism Octant uses is to assign weights to constraints based on their inherent accuracy.

For latency-based constraints, we have observed that the accuracy of constraints from landmarks that have high latency to the target are less trustworthy than those that are nearby. The simple intuition behind this is that the increase in latency is either due to far-away nodes that have a higher probability of traversing through indirect, meandering routes or travel along paths that have high congestion, which often results in constraints that are of relatively little use compared to nearby nodes.

Octant uses a weight system that decreases exponentially with increasing latency, thereby mitigating the effect of high-latency landmarks when lower latency landmarks are present. A weight is associated with each constraint based on the latency between the originating landmark and the target node. When two regions overlap, the weights are added together. In the absence of weights, regions can be combined via union and intersection operations, leading to a discrete solution for a location estimate - the node is either within a region, or lies outside. The introduction of weights changes the implementation. When two regions overlap, Octant determines all possible resulting regions via intersections, and assigns the associated weight to each. The final estimated location region is computed by taking the union of all regions, sorted by weight, such that they exceed a desired weight or region size threshold.

Weights enable Octant to integrate constraints of questionable verity into the system. Recall that, when the resulting area is the empty set, going back and finding the minimal set of constraints that led to an infeasible solution is an NP-complete problem. Weights allow conflicting information to be introduced into the system at little risk of over-constraining the final system and reducing its effectiveness; overaggressive constraints from latency measurements, incorrect zipcode from WHOIS, or misnamed routers in **undns** will not just render the solution down to the empty set. Bad constraints may still impact accuracy if there are no compensating factors, but weights enable Octant to associate a probability measure with regions of space in which a node might lie.

3.5 ITERATIVE REFINEMENT

Localization in the Octant framework can be broken down into two phases. The first is to use accurate and mostly conservative constraints to construct an estimated location region that contains the target with high probability. A second optional phase is to use less accurate and more aggressive constraints to obtain a better estimate of the target location inside the initial estimated region.

In section 3.1, we introduced a scheme by which tight bounds can be established for the negative and positive constraints. While that approach, based on computing the convex hull that includes all inter-landmark measurements, achieves high accuracy in practice, it may sometimes return estimated location regions that are too big and imprecise. The reader will observe that it may be possible to use even more aggressive constraints than those dictated by the discussion so far and achieve smaller estimated location regions. The downside of more aggressive constraints is that they may yield an infeasible system of constraints, where the estimated region collapses down to the empty set. In between these two extremes is a setting at which constraints are set just so that the feasible solution space is below a desired parameter. Iterative refinement is an extension to the basic Octant framework to find this setting.

During the calibration phase, Octant additionally computes for each landmark the interpolated spline that minimizes the square error to the latency-to-distance datapoints of its inter-landmark measurements, as shown with the dashed lines in Figure 4. Opportunistic positive constraints are then derived from the spline by multiplying the spline by a constant δ, while negative constraints are computed by dividing the spline by δ. The value of δ is chosen such that the upper and lower bound contains a given percent of the total number of data points.

Octant initially uses the constraints obtained from the convex hull to compute, typically, a relatively large estimated location area. It then uses this area as a clipping region, which enables it to run through subsequent iterations very quickly, as it can discard all regions that lie outside the initial solution space. The iterative refinement stage then steps through values for δ, recomputing the estimated location area with successively tighter constraints. The process can terminate when the solution space is below a targeted area, is empty, or when a timeout is reached. This optimization enables Octant to determine how aggressively to extract constraints from the network automatically, without hand tuning.

3.6 GEOGRAPHICAL CONSTRAINTS

In addition to constraints extracted from latency measurements, Octant enables any kind of geographical constraint, expressed as arbitrary Bézier-regions, to be integrated into the localization process. In particular, Octant makes it possible to introduce both positive (such as zipcodes from the WHOIS database, zipcodes obtained from other users in the same IP prefix [15]) and negative constraints (such as oceans, deserts, uninhabitable areas) stemming from geography and demographics. Clipping estimated location regions with geographic constraints can significantly improve localization accuracy. Since it is highly unlikely for the system to have landmarks in such areas, negative information is typically not available to rule them out. As a result, estimated regions can extend into oceans. In prior work, which does not permit non-convex regions, the removal of such areas typically requires an ad hoc post-processing step. In contrast, Octant can naturally accommodate such constraints during its estimation process. The application of geographic data, such as ocean boundaries, and demographic data, such as population density maps, can vastly improve precision. Figure 7 shows the city constraints for a target node in Utah, which is otherwise quite difficult to localize precisely due to its distance to all other landmarks and terrain features.

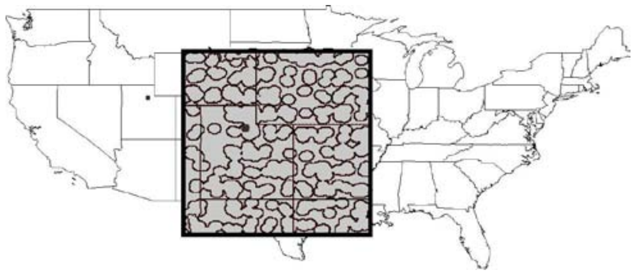


Figure 7: Using the city constraints to localize the planetlab2.flux.utah.edu target can significantly reduce the estimated region size as the gaps between the cities can be removed.

3.7 POINT SELECTION

The Octant approach to localization computes a final estimated localization region which captures the system's best estimate of where the target must lie. Some applications can use such area estimates directly. For instance, web applications might generate content, such as real estate listings, based on the potential zipcodes in which the viewer may be located. Octant can provide the area as either Bézier curve bounded surfaces or an ordered list of coordinates to these applications. Yet many legacy applications, as well as past work, such as GeoPing and GeoTrack, localize targets to a single point. In order to support legacy applications and provide a comparison to previous work, Octant uses a Monte-Carlo algorithm to pick a single point that represents the best estimate of the target's location. The system initially selects thousands of random points that lie within the estimated region. Each point is assigned a weight based on the sum of its distances to the other selected points. After some number of trials, the point with the least weight is chosen as the best estimate of the target's location. This point is guaranteed to be within the estimated location area by definition, even if the area consists of disjoint regions. Ideally, Octant's point selection interface would only serve in a transitional role for legacy application support. We hope

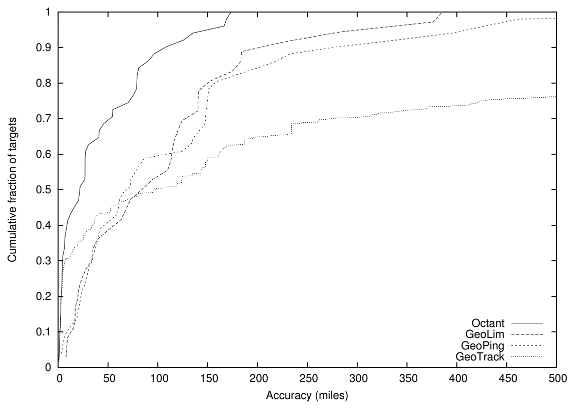


Figure 8: Comparison of the accuracy of different localization techniques in the PlanetLab dataset. Octant achieves significantly greater accuracy than previous work, yielding point estimates for nodes that are substantially closer to the real positions of the targets.

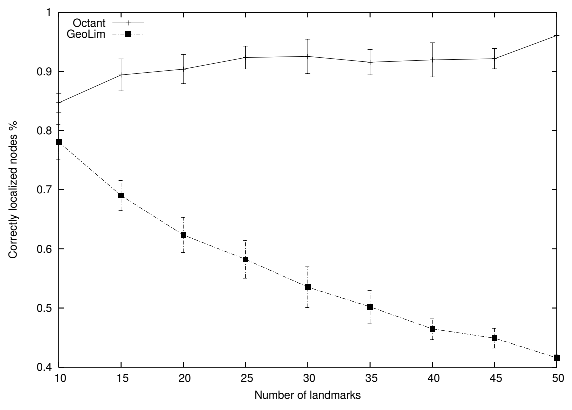


Figure 9: The percentage of targets inside the Octant's location estimate is significantly higher than GeoLim's due to Octant's mechanisms for handling uncertainty of individual landmark's location estimate.

that future applications will be made general enough to take advantage of the extra information in Octant's estimated area.

4 IMPLEMENTATION

The Octant framework for geolocalization is practical, entails low measurement overhead and is computationally inexpensive to execute. The core operations involve the manipulation of Bézier curves, which is a compact representation of curves specified by four control points. Standard libraries support common operations, such as intersection, subtraction, and union on Bézier curves, and implement them efficiently by manipulating only the control points [10]. In addition, Bézier curves are robust to slight inaccuracies in their control points [5].

Our Octant implementation does not depend on having control of the target node, or the intermediate routers between the landmarks and the target. Ideally, the target should respond to probes consistently and quickly. A target behind a slow last mile link, or a slow target that incurs high interrupt and processing latencies for all responses will have its response latency factored into its height, which will then compensate for the node's slower speed. Targets that are inconsistent can pose problems; our current implementation performs 10 ICMP pings and uses the minimum RTT time as the basis for extracted constraints.

Overall, the code consists of 9800 lines of code, whose structure enables it to operate as a third party service, providing geolocalization results given an IP address using only about 50 nodes deployed on the Internet. When a new landmark comes online, it goes through the calibration phase, measures its latencies to other landmarks, and ships its results back to a centralized server. From then on, the landmarks simply perform ping measurements to target IP addresses and report their results to the centralized server. The server performs the localization by combining the constraints. On a 2GHz machine, this operation currently takes a few seconds once the landmarks are calibrated. The system can easily be made decentralized or optimized further, though our focus has been on improving its accuracy rather than its speed, which we find reasonable.

5 EVALUATION

We evaluated Octant using physical latency data collected from a PlanetLab dataset consisting of 51 PlanetLab [7] landmark nodes in North America, as well as a public traceroute server dataset consisting of 53 traceroute servers maintained by various commercial and academic institutions in North America. The latency data for the PlanetLab dataset and the public traceroute servers dataset were collected on Feb 1, 2006 and Sept 18, 2006, respectively, using 10 time-dispersed ICMP ping probes to measure round-trip times. Kernel timestamps were used in the latency measurements to minimize timing errors due to overloaded PlanetLab nodes. To evaluate the efficacy of using secondary landmarks, we also collected the full traceroute information between every landmark and target pair, as well as latency data between the landmarks and intermediate routers. Whenever appropriate, we repeat measurements 10 times, randomizing landmark selection, and plot the standard deviation.

Evaluating the accuracy of a geolocalization system is difficult, because it necessitates a reliable source of IP to location mappings that can be used as the "ground truth." Such an authoritative mapping is surprisingly dif-

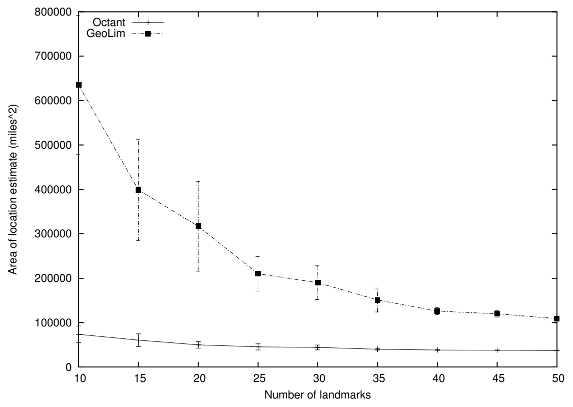


Figure 10: The area of Octant's location estimate is significantly smaller than GeoLim's across all tested number of landmarks.

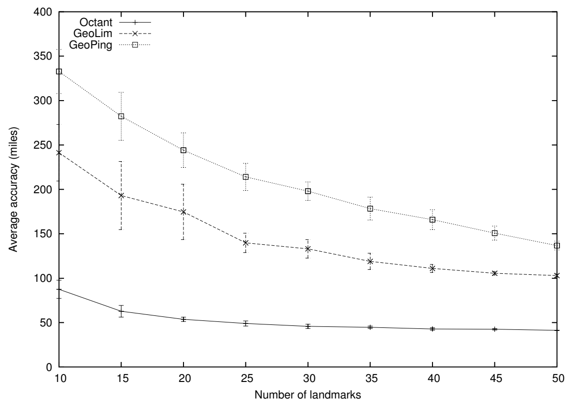


Figure 11: The average distance between a target's physical location and the single point estimate returned for that target. Octant achieves high accuracy with low numbers of landmarks.

ficult to obtain. Our evaluation relies on a total of 104 targets, chosen from the PlanetLab and the traceroute server datasets as described below. We limit our study to North America as the number of PlanetLab nodes on other continents is too few to yield statistically significant results. We vary the number of landmarks inside North America to examine the behavior of the system at lower densities.

In our PlanetLab dataset, nodes serve both as landmarks and targets, following [15, 11]; of course, the node's own position information is not utilized when it is serving as a target. No two hosts in our evaluation reside in the same institution, which rules out simple yet unrealistic and unscalable solutions to geolocalization that rely on having a nearby landmark for every target.

The traceroute servers in the public traceroute server dataset are used only as targets, with 32 PlanetLab nodes serving as landmarks. The individual traceroute server owners specify the location of their traceroute servers as part of the traceroute service. However, these self-provided locations are often erroneous; we eliminate nodes whose reported positions violate speed of light constraints or disagree with a commercial IP geolocation database [2] from consideration.

We first compare Octant with GeoLim, GeoPing, and GeoTrack on the PlanetLab dataset. We evaluate these approaches based on their accuracy and precision, and examine how these two metrics vary as a function of the number of landmarks and average inter-landmark distance. We examine accuracy in terms of two metrics: one is the distance between the single point estimate returned by these geolocalization techniques and the physical location of the node, while the other is the percent of nodes whose real-world locations reside within the estimated location area. The former metric allows us to compare Octant to previous systems, such as GeoPing and GeoTrack, that compute only a point estimate, and evaluates how well these systems perform for legacy applications that rely on a single point position. The latter, applicable only to recent geolocalization systems including Octant and GeoLim and proposed by GeoLim [11], evaluates how well area-based approaches perform. Note that comparisons using the latter metric need to be accompanied by measurements on the size of the estimated area (otherwise a simple solution containing the globe will always locate the node accurately), which we also provide.

Figure 8 shows the accuracy of different geolocalization techniques by plotting the CDF of the distance between the position estimate and the physical location of a node. Octant significantly outperforms the other three techniques across the entire set of targets. The median accuracy of Octant is 22 miles, compared to median accuracy of 89 miles for GeoLim, 68 miles for GeoPing and 97 miles for GeoTrack. GeoLim was not able to localize approximately 30% of the targets, as its over-aggressive constraint extraction leads to empty regions. Even the worst case under Octant is significantly lower than the worst cases encountered with other techniques. The worst case under Octant, GeoLim, GeoPing and GeoTrack are 173, 385, 1071, and 2709 miles, respectively.

A median error of 22 miles is difficult to achieve based solely on constraints obtained online from uncoordinated and unsynchronized hosts, in the presence of routing anomalies and non-great circle routes. As a point of comparison, if all hosts on the Internet were connected via point-to-point fiber links that followed great-circle paths, host clocks were synchronized, and there were no routing anomalies, achieving such error bounds using packet timings would require timing accuracy that could accu-

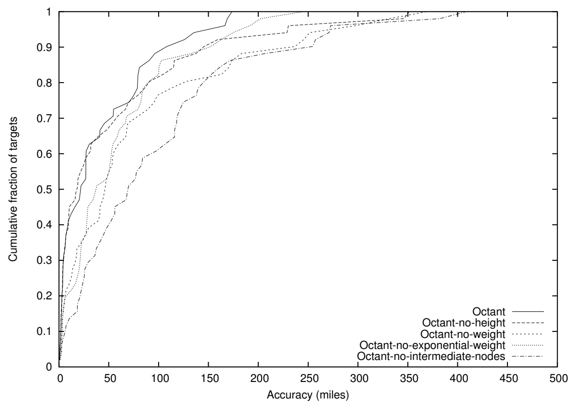


Figure 12: The contributions of individual optimizations used in Octant to geolocalization accuracy. The use of intermediate nodes provides the largest improvement to system accuracy.

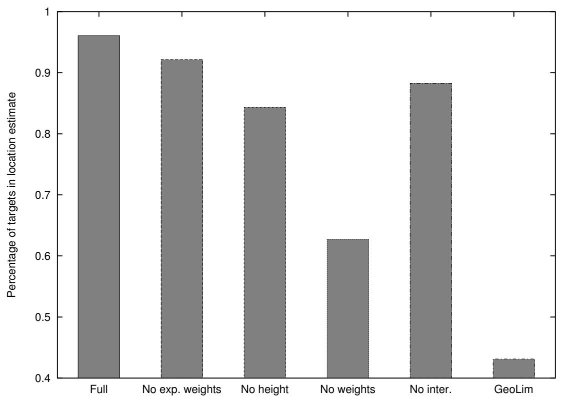


Figure 13: The percentage of targets located on average within their estimated location areas for Octant, Octant without various optimizations, and GeoLim.

rately distinguish a delay of $22 * 1.6/(2/3 * 300000) = 170$ microseconds.

In a typical deployment, the number of landmarks used to localize a target is often constrained by physical availability, and an implementation may not be able to use all landmarks in the localization of all targets due to load limits, node failures, or other network management constraints. We evaluate Octant's performance as a function of the number of landmarks used to localize targets, and compare to GeoLim, the only other region-based geolocalization system. Figure 9 shows the number of nodes that were located successfully; that is, their physical locations were inside the estimated location region returned by Octant. Three findings emerge from the plot. First, the percentage of nodes that are successfully localized is quite high for Octant, averaging more than 90% when the number of landmarks exceeds 15. Second, the accuracy of the Octant approach remains flat or improves slightly with increasing numbers of landmarks. Using 15 landmarks yields results that are almost as good as using all 50, and adding more landmarks does not hurt performance. Finally, the accuracy of the GeoLim approach is high for low numbers of landmarks, and drops as more landmarks are introduced. This surprising behavior is due to overaggressive extraction of constraints in GeoLim; as the number of landmarks increases, the chances that a "bad" node, whose network connection yields an unexpected delay, will introduce an over-constraint grows. When there are too many conflicting constraints, GeoLim yields the empty set as its location estimate, whereas the weighted combination of constraints enables Octant to avoid these pitfalls. With all 50 landmarks, GeoLim returns the empty set for more than 29% of the targets.

The preceding evaluation, which examined the percentage of nodes whose physical locations lie inside their estimated location region, needs to be coupled with an examination of the size of the estimated location regions to put the accuracy of the systems in context. Figure 10 shows the area of the estimated location region for GeoLim and Octant. The size of the geolocalization region is quite small for Octant at 10 landmarks, and remains the same or slowly decreases with additional landmarks. For small numbers of landmarks, GeoLim returns substantially larger areas that are a factor of 6 bigger than Octant's, which explains its ability to localize about 80% of the nodes with 10 landmarks. Adding more landmarks refines GeoLim's location estimates, though even at 50 landmarks, GeoLim's location estimates are twice the size of Octant's. Octant is able to keep the region small via its careful extraction of constraints and use of negative information.

We next examine the absolute error in legacy position estimates based on a single point. Figure 11 plots the average distance between a target's physical location and the single point estimate returned for that target. Octant consistently achieves high accuracy, even with landmarks as few as 15. In contrast, GeoLim and GeoPing exhibit performance that degrades as the number of landmarks decreases and their distribution throughout the space becomes more sparse. Octant's performance as the number of landmarks decreases mostly stems from its ability to use routers inside the network as secondary landmarks. Octant's average error is significantly less than both GeoPing and GeoLim even when Octant uses a fifth of the landmarks as the other schemes.

To provide insight into Octant's accuracy, we examine its performance as we disable various optimizations. We examine the individual contribution of each of our opti-

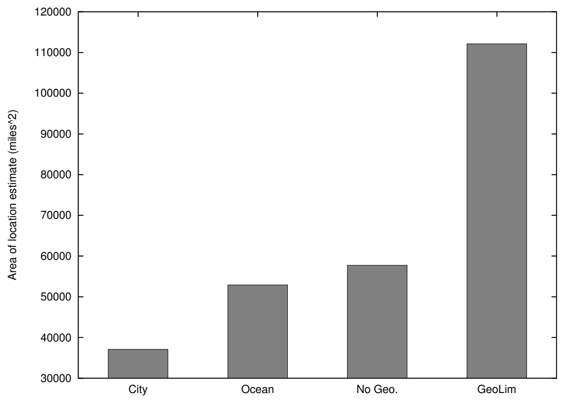


Figure 14: The area of the location estimate for Octant with demographic and geographic constraints. The use of these exogenous constraints substantially reduce the size of the estimated area.

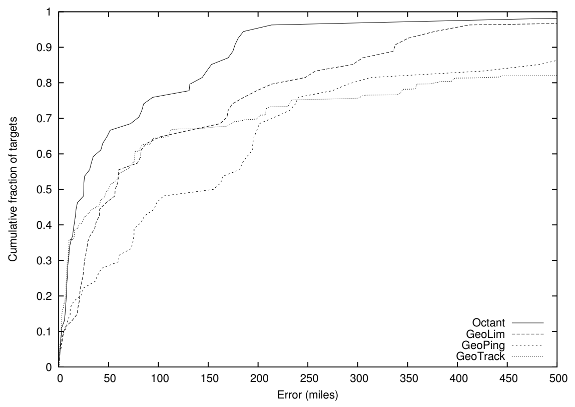


Figure 15: The accuracy of different localization techniques on the public traceroute servers dataset show very similar results to those from the PlanetLab dataset, with Octant yielding point estimates that are significant closer to the real positions of the targets.

mizations, namely heights, weights, exponential weights and intermediate nodes, by turning off each one in turn and comparing their accuracy with that of the complete Octant system. Figure 12 shows the resulting CDFs. The largest improvement to system accuracy is due to the use of intermediate nodes, which significantly increases the number of usable landmarks in the system. **Undns** is useful approximately half the time, but transitive localization is critical to precise localization.

Figures 13 and 14 provide further insight into the impact of various optimizations on Octant's accuracy and precision. Figure 13 plots the percentage of nodes localized with each of the optimizations turned off. NoInter refers to Octant with localization through secondary landmarks inside the network turned off, NoWeights uses no weights associated with constraints, NoHeight disables the last hop delay approximation, NoExpWeights uses weights but all constraints carry equal weights. The distinction between NoWeights and NoExpWeights is subtle but important. In NoWeights, the estimated location of the target is the intersection of all available constraints. In contrast, NoExpWeights estimates the location of the target as the union of all regions above a certain weight threshold. The effects of a limited number of incorrect constraints can be mitigated by trading off precision, as chosen by the threshold value. The largest contribution in improving accuracy, approximately 33%, stems from the use of weights. GeoLim is less accurate than all the different Octant configurations, even though its location estimates are significantly larger.

The use of geographical constraints in Octant significantly reduces the size of the location estimates. Figure 14 shows the size of the location estimates in square miles for Octant with the population density ("cities") constraint which introduces clipping areas into location estimates weighted by the density of the population inside that region, with the oceans constraint which clips oceans from the solution space, and without any geographic constraints, as well as the location estimate area for GeoLim. The use of either geographical constraint alone makes a significant improvement in the location estimate, improving the precision of the estimates. Combined, these geographical estimates greatly improve the fidelity of the location estimates returned by Octant.

The results from our public traceroute servers dataset which includes a mixture of smaller commercial organizations and academic institutions are very similar to those from our PlanetLab dataset. Figure 15 shows the CDF of the distance between the position estimate and the physical location of a node. Octant again outperforms the other three techniques across the entire set of targets, with a median localization error of 25 miles, compared to 56 miles for GeoLim, 155 miles for GeoPing and 50 miles for GeoTrack. The significant decrease in accuracy for GeoPing is likely due to the reduction of landmarks from 51 in the PlanetLab dataset to 32 in the traceroute servers dataset, as GeoPing is the most sensitive technique to the number of available landmarks.

These results show that Octant achieves high accuracy in its point estimate for a target, high probability that its location estimate will contain the target, and high precision as indicated by the size of its location estimate area. Overall, Octant can locate the median node to a point within 22 miles of its physical position, or to a tight area (factor of two smaller than previous work) that contains the physical location with 90% probability. Octant requires very few landmarks to be effective; as few as 20 landmarks can achieve approximately the same accuracy as from using all 50 landmarks.

6 CONCLUSIONS

Determining the geographic location of Internet hosts is an intrinsically useful basic building block. Since there are no existing standardized protocols for discovering the physical location of endpoints, we must rely on techniques that can extract location information from network measurements.

In this paper, we have outlined Octant, a general, comprehensive framework for representing network locations for nodes, extracting constraints on where nodes may or may not be located, and combining these constraints to compute small location estimates that bound the possible location of target nodes with high probability. Octant represents node position and regions precisely using Bézier-bounded regions that can admit any kind of constraints, makes use of both positive and negative information to aggressively reduce the estimated region size, and can effectively reason in the presence of uncertainty and erroneous constraints. It utilizes a number of techniques to extract fine-grain information from end-to-end latency measurements on the Internet.

We have evaluated our system using measurements from PlanetLab hosts as well as public traceroute servers and found that Octant is surprisingly accurate and effective. The framework can localize the median node to within 22 miles of its actual position. The evaluation also indicates that Octant can localize a target to a region that is less than half the size of previous approaches and contains the target with much higher probability than the larger region. Octant enables network operators to determine, with high confidence, the position of nodes given simply latency measurements, which in turn enables new location-aware services.

ACKNOWLEDGMENTS

We would like to thank our shepherd, T.S. Eugene Ng, and the anonymous reviewers for many helpful comments and suggestions [4].

References

[1] IP to Latitude/Longitude Server, University of Illinois. http://cello.cs.uiuc.edu/cgi-bin/slamm/ip2ll.

[2] IP2Location. http://www.ip2location.com.

[3] Quova. http://www.quova.com.

[4] Visualware Inc. http://www.visualroute.com.

[5] Assaf, D. *The Sensitivity of Spline Functions on Triangulations to Vertex Perturbation*. PhD thesis, Vanderbilt University, May 1998.

[6] Bahl, P., and Padmanabhan, V. RADAR: An In-Building RF-Based User Location and Tracking System. In *INFOCOM* (Tel Aviv, Israel, March 2000).

[7] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. Operating System Support for Planetary-Scale Network Services. In *Networked Systems Design and Implementation* (San Francisco, CA, March 2004).

[8] Cao, J., Davis, D., Wiel, S., and Yu, B. Time-varying Network Tomography. *American Statistical Association 95* (2000).

[9] Dabek, F., Cox, R., Kaashoek, F., and Morris, R. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM* (Portland, OR, August 2004).

[10] Farin, G. *Curves and Surfaces for Computer Aided Geometric Design: A Practical Guide*. Academic Press, 1988.

[11] Gueye, B., Ziviani, A., Crovella, M., and Fdida, S. Constraint-Based Geolocation of Internet Hosts. In *Internet Measurement Conference* (Taormina, Sicily, Italy, October 2004).

[12] Guha, S., Murty, R., and Sirer, E. Sextant: A Unified Framework for Node and Event Localization in Sensor Networks. In *Mobihoc* (Urbana-Champaign, IL, May 2005).

[13] Katz-Bassett, E., John, J., Krishnamurthy, A., Wetherall, D., Anderson, T., and Chawathe, Y. Towards IP Geolocation using Delay and Topology Measurements. In *Internet Measurement Conference* (Rio de Janeiro, Brazil, October 2006).

[14] Moore, D., Periakaruppan, R., and Donohoe, J. Where in the World is netgeo.caida.org? In *INET2000 Poster* (Yokohama, Japan, July 2000).

[15] Padmanabhan, V., and Subramanian, L. An Investigation of Geographic Mapping Techniques for Internet Hosts. In *SIGCOMM* (San Diego, CA, August 2001).

[16] Periakaruppan, R., and Nemeth, E. GTrace - A Graphical Traceroute Tool. In *LISA* (Seattle, WA, November 1999).

[17] Savage, S., Collins, A., and Hoffman, E. The End-to-End Effects of Internet Path Selection. In *SIGCOMM* (Cambridge, MA, September 1999).

[18] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP Topologies with Rocketfuel. In *SIGCOMM* (Pittsburgh, PA, August 2002).

[19] Vardi, Y. Network Tomography: Estimating Source-Destination Traffic Intensities from Link Data. *American Statistical Association 91* (1996).

Notes

[1] In this context, *accuracy* refers to the distance between the computed point estimate and the actual location of the target. In contrast, *precision* refers to the size of the region in which a target is estimated to lie.

[2] An octant is a navigational instrument used to obtain fixes.

[3] Note that this difference might embody some additional transmission delays stemming from the use of indirect paths. We expand on this in the next section.

[4] This work was supported in part by AF-TRUST (Air Force Team for Research in Ubiquitous Secure Technology for GIG/NCES), which receives support from the DAF Air Force Office of Scientific Research (#FA9550-06-1-0244) and the following organizations: the National Science Foundation (CCF-0424422) Cisco, British Telecom, ESCHER, HP, IBM, iCAST, Intel, Microsoft, ORNL, Pirelli, Qualcomm, Sun, Symantec, Telecom Italia, and United Technologies.

dFence: Transparent Network-based Denial of Service Mitigation

Ajay Mahimkar, Jasraj Dange, Vitaly Shmatikov, Harrick Vin, Yin Zhang
Department of Computer Sciences, The University of Texas at Austin
{mahimkar,ja raj, hma , i , ha a

Abstract

Denial of service (DoS) attacks are a growing threat to the availability of Internet services. We present dFence, a novel network-based defense system for mitigating DoS attacks. The main thesis of dFence is *complete transparency* to the existing Internet infrastructure with no software modifications at either routers, or the end hosts. dFence dynamically introduces special-purpose middlebox devices into the data paths of the hosts under attack. By intercepting both directions of IP traffic (to and from attacked hosts) and applying stateful defense policies, dFence middleboxes effectively mitigate a broad range of spoofed and unspoofed attacks. We describe the architecture of the dFence middlebox, mechanisms for on-demand introduction and removal, and DoS mitigation policies, including defenses against DoS attacks on the middlebox itself. We evaluate our prototype implementation based on Intel IXP network processors.

1 Introduction

Denial of service (DoS) attacks pose a significant threat to the reliability and availability of Internet services. Consequently, they are emerging as the weapon of choice for hackers, cyber-extortionists [24], and even terrorists [26]. The widespread availability of attack tools [3] makes it relatively easy even for "script kiddies" to mount significant denial of service attacks.

Our goal is to design and build a transparent network-based defense system capable of mitigating a broad range of large-scale, distributed denial of service attacks directly inside the network, without requiring software modification at either routers, or end hosts. Such a system can be deployed by Internet service providers (ISPs) in *today's* Internet, providing on-demand protection to customers, including those who operate legacy servers, only when they experience an actual attack. It can also serve as a general platform on which new security services and defense mechanisms can be deployed at a low cost, with a single installation protecting a large number of customers.

The problem of detection and mitigation of denial of service attacks has received considerable attention. Despite a large body of research literature and availability of commercial products, effective protection against denial of service has remained elusive. There are several plausible reasons for this. Most of the proposed solutions require software modification at either routers, or end hosts, or both. This means that the defense is not transparent to the Internet service providers and/or their customers.

Solutions that are not directly compatible with the existing TCP/IP implementations and routing software are likely to face unsurmountable deployment obstacles. Even SYN cookies, which are backward-compatible and part of standard Linux and FreeBSD distributions, are not used by the vast majority of users because they are turned off by default. At the same time, waiting until the Internet is re-engineered to provide better resistance against denial of service is not a feasible option for users who need immediate protection.

Arguably, the main challenge in DoS research today is not only coming up with new defense methods, but also finding an effective way to deploy both existing and new defenses with no changes to the installed software base, and without any performance cost when denial of service activity is not happening.

This problem is not unique to denial of service. Many network attacks are relatively rare events: a given end host may experience a denial of service attack once every few months, or be exposed to a new worm once a year. Therefore, there is little incentive for the end host operator to deploy an expensive protection system or to modify the existing software, especially if the change affects normal network performance. Of course, *not* deploying a defense can have catastrophic consequences when the attack does happen. We solve this conundrum by providing technological support for a "group insurance service" that an ISP can offer its customers: an on-demand defense that turns on only when the customer is actually experiencing an attack, and otherwise has no impact on the network operation.

Our contributions. We present *dFence*, a novel DoS mitigation system based on a small number of special-purpose "middlebox" devices located in the middle of the network. The main features of dFence are:

- **Transparency:** dFence is fully transparent to the end hosts, requires no modifications of client or server software, and enables protection of legacy systems. Protected hosts need not even be aware that the system is in operation.

- **Compatibility with routing infrastructure:** dFence employs standard intra-domain routing and tunneling mechanisms for traffic interception. It requires no changes to the existing router software, and is thus incrementally deployable by Internet service providers.
- **On-demand invocation:** dFence middleboxes are dynamically introduced into the data path of network connections whose destinations are experiencing denial of service, and removed when the attack subsides. The small performance cost of filtering is paid *only* by the attacked hosts, and only for the duration of the attack.
- **Scalability:** The dynamic nature of dFence allows ISPs to multiplex the same defense infrastructure to protect a large number of customers (who are not all under attack *at the same time*), and thus more efficiently utilize their network and computing resources.
- **Minimal impact on legitimate connections:** Each dFence middlebox manages the state of active, legitimate connections to the customers who are simultaneously under attack. Malicious flows do not occupy any memory at the middlebox. Therefore, legitimate flows can be processed with very low latency cost. The cost for the flows to the destinations *not* experiencing an attack is zero.
- **Versatility:** Because dFence middleboxes dynamically intercept *both* directions of TCP connections to DoS-affected hosts, they can apply stateful mitigation policies to defend against the entire spectrum of DoS attacks.
- **Economic incentive:** We envision dFence middleboxes being deployed within a single ISP. The ISP can then charge a premium to customers who subscribe for a dFence-based "insurance service." dFence middleboxes are turned on only when one or more paying customers are experiencing an attack. As more customers subscribe to the service, the ISP can incrementally scale up the deployment.

System architecture. Figure 1 depicts the overall system architecture. The two guiding design principles behind dFence are *dynamic introduction* and *stateful mitigation*. We implement dynamic introduction by using intra-domain routing and tunneling mechanisms to transparently insert dFence middleboxes into the data path of traffic destined to hosts experiencing a DoS attack. This is done *only* when DoS activity is detected in the network. Due to dynamic introduction, our solution has zero impact on normal network operations, and can be deployed incrementally.

The middleboxes intercept both directions of network traffic (to and from attacked hosts), which enables many

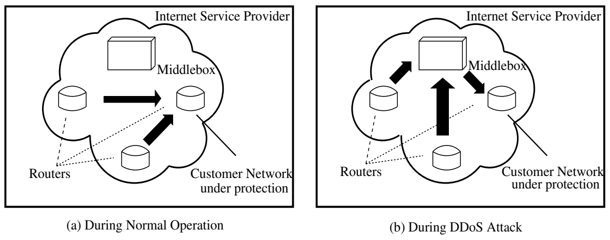

Internet Service Provider
Middlebox
Routers
Customer Network under protection
(a) During Normal Operation
Internet Service Provider
Middlebox
Routers
Customer Network under protection
(b) During DDoS Attack

Figure 1: dFence Architecture: (a) During normal operation, the ingress routers forward the traffic towards the corresponding egress routers. (b) Under a large-scale DoS attack, traffic is re-directed via the middlebox. The middlebox applies mitigation policies and filters out illegitimate traffic.

mitigation techniques that were previously considered unsuitable for network-based defenses. The main technical novelty is the network-based implementation of defenses that previously required modifications to server or client software. For example, "outsourcing" SYN cookie generation to dFence middleboxes enables us to protect legacy end hosts whose TCP/IP implementations do not support SYN cookies. Other mitigation techniques include defenses against spoofed and unspoofed data floods, against clients opening and abandoning a large number of connections, and against distributed botnet attacks.

Key challenges. The combination of transparent on-demand defense, two-way traffic interception, and stateful mitigation presents several interesting challenges: (i) how to deal with middlebox transitions, *i.e.*, how to introduce and and remove middleboxes on selected data paths; (ii) how to dynamically bootstrap, manage, and remove connection state at the middleboxes; (iii) how to handle network behavior such as route changes and failures of network elements; and (iv) how to handle overload conditions and DoS attacks on the middleboxes themselves.

We present practical solutions to all of these challenges. Our main contribution is a careful integration of several network mechanisms leading to a *completely transparent, scalable and effective DoS mitigation system*. We evaluate our design using a prototype implementation based on Intel's IXP2400 network processors, and demonstrate that mitigation of a broad range of DoS attacks, including spoofed SYN floods and unspoofed data floods, can be achieved with minimal performance degradation. We also note that the focus of this paper is on mitigation, rather than detection of denial of service activity.

Organization. The rest of the paper is organized as follows. In Section 2, we describe the mechanisms for dynamically introducing dFence middleboxes in the data path of attack traffic. Section 3 describes the architecture of the dFence middleboxes and the mitigation policies that defend against a broad range of spoofed and unspoofed attacks. Our prototype implementation and its performance evaluation are presented in Sections 4 and 5, respectively. Section 6 outlines the related work. Section 7 summarizes our contributions.

2 Transparent Middlebox Invocation

A key design principle for dFence middleboxes is complete transparency to the end hosts. This is achieved through *dynamic invocation* of middleboxes by standard intra-domain routing mechanisms and tunneling. A few dFence middleboxes are introduced into the network to provide focused protection to the subset of the end hosts that are currently experiencing a DoS attack. Protected hosts do not need to modify their software, nor access the network through special overlay points, nor set up IP tunnels, nor even be aware that dFence middleboxes have been deployed inside their ISP.

Dynamic middlebox invocation is critical for deployability because it ensures that during peace time (*i.e.*, when there is no ongoing DDoS activity) customer traffic does not have to pay the penalty of triangular routing through the middleboxes. Dynamic middlebox invocation is also important for the defense system itself because it focuses all defense resources only on the connections whose destinations are under attack, leaving other customers unaffected. The defense system can thus benefit from statistical multiplexing and potentially protect many more customer networks with the same available resources.

Combining dynamic middlebox invocation with stateful attack mitigation raises several technical challenges.

- *Bidirectional traffic interception.* Many of our mitigation policies require the defense system to capture both directions of customer traffic. For example, to protect against spoofed data floods from remote hosts to a customer network under protection, we maintain a Connection table to summarize on-going TCP connections. Bidirectional traffic interception is difficult in general because Internet routing is destination-based by default; intercepting traffic from the customer network, however, requires the ability to perform *source*-based routing.

- *Flow pinning.* In addition to intercepting both directions of protected traffic, stateful mitigation also requires that both directions pass through the *same* middlebox (where the state of the connection is maintained), even after routing changes have caused the intercepted traffic to use different ingress points. This requires a mechanism for pinning each flow, defined by the TCP/IP packet header, to a particular middlebox. Flow pinning also provides security against an attacker who attempts to disrupt external routing while launching an attack.

  We use a simple hash-based flow pinning method. Each flow is associated with a *home* middlebox, whose identity is determined by a hash $h(f)$ of the flow identifier f. The flow identifier consists of source and destination IP addresses and port numbers in the TCP/IP packet header. If the flow is intercepted by a *foreign* middlebox, it is simply forwarded to the home middlebox, achieving reasonable load balancing of flows across the middleboxes. In the future, we plan to investigate more sophisticated flow pinning mechanisms.

- *Dynamic state management.* Dynamic middlebox invocation also poses interesting challenges for state management. The first question is how to handle existing connections when the middlebox is inserted into the data path. For instance, our policy for defending against spoofed data flooding can drop a data packet if it is not in the Bloom filter summary of ongoing connections. But an existing legitimate connection may not be present in the Bloom filter if it had been established before the middlebox was introduced. We would like to minimize the number of packets dropped for such connections.

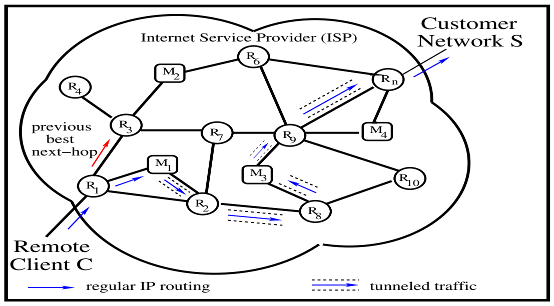
  Besides filtering, some of our advanced mitigation policies perform operations that change the content of the traffic. For example, the "outsourced" SYN cookie policy requires the splicing of TCP connections and sequence number translation to be performed at the middlebox. What happens to the spliced connections when the middlebox is removed from the data path?

- *Middlebox failure recovery.* A dFence middlebox may fail due to software or hardware errors, or traffic overload. Protecting the overall defense system from middlebox failures is an important challenge.

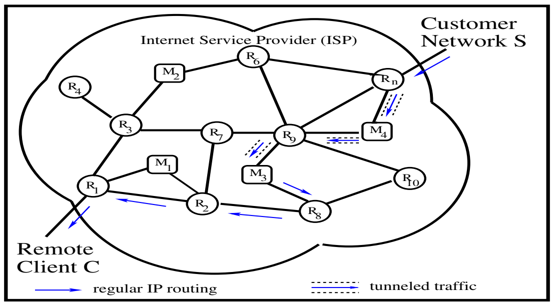
Our solution is based on standard BGP/IGP routing, tunneling, and policy-based routing (PBR) [7], which is available in almost all existing routers. Therefore, our solution is easy to deploy in today's Internet. In addition, we implement a simple hash-based mechanism for pinning each individual connection to the same *home* middlebox. This ensures that both directions of the connection traverses the same middlebox (even in the presence of route changes that may result in different ingress points). Below we present our approach for bidirectional traffic interception, dynamic state management, middlebox failure recovery and load balancing.

2.1 Dynamic Traffic Interception

While there exist several academic and industrial solutions for traffic interception [1, 8, 11], none of them, to the best of our knowledge, can simultaneously (i) introduce middleboxes dynamically, (ii) intercept both inbound and outbound traffic, and (iii) ensure that both directions of a connection go through the same home middlebox. As a result, no single existing technique is sufficient for stateful attack mitigation. It is possible, however, to use existing techniques to substitute some of the individual components of our solution (*e.g.*, our mechanism for inbound traffic interception).



(a) Inbound Traffic Interception



(b) Outbound Traffic Interception

Figure 2: Traffic interception at middleboxes using BGP/IGP, tunneling and policy-based routing

2.1.1 Inbound traffic interception

In our prototype implementation, we use iBGP and tunneling to intercept inbound traffic (alternatives include mechanisms such as MPLS tunneling). To intercept all traffic inbound to some customer network S, the dFence middleboxes send iBGP updates advertising a route to S with the highest local preference to all BGP peers in the local AS. As a result, the middleboxes become the preferred egress points for all traffic destined to S. At each ingress router, IGP selects the closest middlebox and updates the forwarding tables appropriately.

To enable the packets to reach S after they have been filtered by the middlebox, dFence configures a tunnel from the middlebox to the real egress router associated with S. The tunnel can be set up using any available mechanism; in our prototype, we use IP-IP tunnels. The egress router specifies two ACLs: (a) *ACL-to-S* is defined on the router's internal interface (connecting it to the rest of the ISP) and intended for traffic going towards S; (b) *ACL-from-S* is defined on the external interface connecting it to the customer network and intended for traffic arriving from S.

The journey of an incoming packet typically consists of the following three steps, as illustrated in Figure 2(a).

1. *Go to one of the middleboxes:* IGP selects the middlebox M_1, which is the closest middlebox to the ingress point. If M_1 is the home middlebox for this flow, the next step is skipped; otherwise M_1 needs to forward the packet to its home middlebox.
2. *Flow pinning:* The packet is tunneled from the foreign middlebox M_1 to the home middlebox M_3. The identity of the home middlebox is determined by the hash value of the flow identifier. The home middlebox M_3 then applies mitigation policies described in section 3.2 to process the packet.
3. *Go to the real egress router:* After the middleboxes advertised routes to S, the intermediate routers' forwarding tables point towards middleboxes for all packets whose destination is S. To avoid routing loops, we tunnel the packet from the home middlebox M_3 to the true egress router R_n. When R_n receives the tunneled packet, it decapsulates the packet and, because it matches *ACL-to-S*, forwards it to the customer network S.

Observe that the traffic arriving on the *external* interfaces of R_n (other than the interface connecting it to S) will be first re-routed to the middlebox for filtering, because the middleboxes' iBGP route advertisements change the forwarding table at R_n, too.

2.1.2 Outbound traffic interception

We use policy-based routing (PBR) [7] to intercept outbound traffic originating from the customer network S to a remote host. PBR allows flexible specification of routing policies for certain types of packets, including ACLs (access control lists) that identify classes of traffic based on common packet header fields. In our context, we use PBR to forward all traffic that matches *ACL-from-S* to a dFence middlebox through a preconfigured tunnel.

The journey of an outbound packet consists of the following three steps, as illustrated in Figure 2(b).

1. *Go to one of the middleboxes:* When the egress router R_n receives a packet from S, the flow definition matches *ACL-from-S* and so the packet is forwarded to middlebox M_4 through its preconfigured tunnel interface.
2. *Flow pinning:* M_4 forwards packets to the home middlebox M_3, determined by the flow pinning hash function. The hash function is selected in such a way that exchanging source and destination fields does not affect the hash value, for example:

$$h_1(src\_addr, src\_port) \oplus h_2(dst\_addr, dst\_port)$$

   where h_1 and h_2 are two independent hash functions. This ensures that both directions of the same connection have the same home middlebox (M_3 in our example)
3. *Go towards the egress router*: Regular IP routing is used to send the packet to its egress router R_1.

2.2 Dynamic State Management

The main challenge in dynamic middlebox invocation is the need to gracefully handle existing connections upon the introduction or removal of a middlebox. Our basic solution is to add *grace periods* after the introduction or before the removal of the middlebox. During the grace

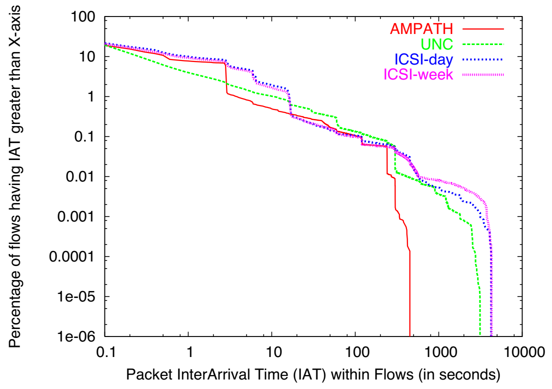


Figure 3: Packet inter-arrival time (IAT) within flows

period, the middlebox continues to serve all existing connections while it is preparing to establish or time out its state.

After the middlebox has been introduced into the data path, it spends T_b seconds (*state bootstrap interval*) bootstrapping its state. After the decision has been made by the network operator to remove the middlebox, the middlebox stays in the data path for another T_r seconds (*state removal interval*) before being completely removed.

1. *State bootstrapping:* During interval T_b, the middlebox establishes state for the existing connections between clients and the server and/or customer network which is being protected. An existing connection is considered legitimate if the middlebox sees both inbound and outbound traffic on it during the bootstrap period. The list of connections is maintained using the Connection table data structure, described in detail in Section 3.1.

2. *State removal:* After the removal decision has been made, the middlebox can be removed right away if no currently active mitigation policy involves modifications to individual packet content. Some mitigation policies, however, such as "outsourced" SYN cookie generation cause the middlebox to actively modify packet headers (*e.g.*, by sequence number translation). For these policies, the middlebox cannot be removed right away because the existing connections can become invalid without the translations performed by the middlebox. Therefore, the middlebox needs to remain in the data path during interval T_r and continue to serve the ongoing connections. No policies are applied on any new connections and they are directly forwarded to their destinations.

The value of the state bootstrap interval T_b is important. If T_b is too long, then the attacker can cause severe damage while the middlebox is being bootstrapped. If T_b is too short, then many existing connections may either get terminated, or suffer poor performance. Trace analysis based on several large datasets shows that the vast majority (99%) of all connections have packet interval times that are quite small, on the order of a few seconds (see Figure 3). Hence, T_b can be set to 5 or 10 seconds. This means that, within a short bootstrap interval, the state for the vast majority of existing connections can be established, and only a handful of legitimate users (those who were idle during T_b) will have to re-establish connections.

It might appear that the bootstrap time T_b provides an opportunity for an attacker to overload the middlebox itself. This is not the case because connection state is maintained at the middlebox only for unspoofed connections which comply with traffic control measures.

The decision to introduce a middlebox can be made by the customer networks under protection (*e.g.*, when they observe too much inbound traffic), or through some network-based DoS detection system. Since our primary focus in this paper is on attack mitigation, we do not discuss attack detection here.

The value of the removal interval T_r can be pre-specified (it must be sufficiently long to allow for normal termination of all ongoing connections), or it can be adaptive based on the number of connections going through the middlebox. Compared with T_b, the choice of T_r is less critical because it primarily affects the amount of additional work that the middlebox performs and has little impact on the safety of customer networks.

The decision to remove a middlebox can only be made by the middlebox itself (or by all middleboxes collectively). Unlike middlebox introduction, middlebox removal cannot be decided by the customer networks—if the defense system is effective, then the customers should not be able to tell whether the DoS attack ceased, or the attack is still in progress. Therefore, the middleboxes need to continuously profile the (unfiltered) traffic and decide whether the ongoing attack has subsided.

2.3 Failure Recovery and Load Balancing

A middlebox can fail for a variety of reasons, such as power outage, hardware malfunction, software errors, network outages, and so on. It can also fall victim to a DoS attack itself, even though our middlebox architecture is explicitly hardened against this possibility. When a middlebox fails, it is crucial that a different middlebox (or the same middlebox after rebooting) take over the management of all connections whose flow identifiers have been pinned to the failed middlebox. To make this transition as smooth as possible, the middlebox must offload its state to a different middlebox as soon as overloading is detected. Therefore, *graceful flow migration* is the key component of both failure recovery and load balancing. We outline how it can be achieved in the dFence mitigation system.

Recall that each flow identifier f is pinned to its home middlebox by a hash function $h(f)$. To avoid changing the hash function (which may be implemented in hardware and difficult to modify), we introduce one level of indirection. All middleboxes in our system will agree on a global *home middlebox table* $HM[0..n-1]$ (*e.g.*,

$n = 1024$). Each middlebox is responsible for a subset of entries in this table, or, more precisely, for all flows whose identifiers hash to this subset. The global *HM* table can be maintained either centrally, or through a distributed agreement protocol among all the middleboxes. The hash function h can be chosen so that it maps flow identifier f to $\{0, 1, \cdots, n-1\}$. The home middlebox for flow f is simply $HM[h(f)]$. This enables graceful flow migration for failure recovery and load balancing.

Failure recovery. All middleboxes are pre-arranged to form a logical ring R. For each middlebox M_i, its clockwise next-hop neighbor in R (denoted by $R.\mathrm{next}(M_i)$) is the designated backup for M_i and will take over the flows managed by M_i should M_i fail. Suppose for some entry e the original home middlebox $M_1 = HM[e]$ failed. $M_2 = R.\mathrm{next}(M_1)$ then becomes the new $HM[e]$ and starts the bootstrap interval T_b, during which it bootstraps (as described in section 2.2) the state of all ongoing connections whose flows are hashed to e. The same procedure can be repeated to handle multiple failures. For example, if both M_1 and M_2 failed, then $M_3 = R.\mathrm{next}(M_2)$ becomes the new $HM[e]$. Note that unreachable middleboxes due to network partition can be handled in the same way as failures. The only additional processing required for coping with network partition is to resolve the inconsistency in the *HM* tables maintained by different partitions after the network becomes connected again.

Load balancing. Load balancing can be handled in a similar manner. Suppose M_1 is overloaded and wants to offload all flows that are hashed to entry e to a less loaded middlebox M_2. All it needs to do is to update the global home middlebox table so that $HM[e] = M_2$. M_2 then spends the T_b period to bootstrap its state for flows that are mapped to e. Note that during the state bootstrap interval, instead of blindly letting through every flow that M_2 has no state for, M_2 has the option of forwarding such flows to M_1. This can make flow migration more graceful, especially when M_1 has been applying traffic-modifying mitigation policies such as SYN cookies and sequence number translation.

3 Middlebox Design

3.1 Overview

dFence is based on diverting traffic to special-purpose *middleboxes* as soon as denial of service activity is detected. Each middlebox is responsible for protecting TCP connections to some or all of the attacked destinations. To effectively distinguish between benign and malicious traffic, the middlebox maintains partial TCP state for both directions of the intercepted connections, but does not buffer any packets. Because mitigation is performed entirely within the middlebox and traffic redirection is achieved using standard intra-domain routing and tunneling mechanisms, dFence does not require any software modification at either routers, or the end hosts. We focus on TCP-based attacks, but UDP, ICMP or DNS attacks could be handled in a similar fashion.

TCP connection management. TCP connections managed by a dFence middlebox include pre-existing connections that had been established before the middlebox was introduced into the data path (the middlebox acquires these connections during the bootstrap period—see section 2.2), and those established after the middlebox became active. The latter connections are *spliced* to enable anti-spoofing defenses. Splicing performed by the middlebox is very simple and limited to translation of sequence numbers.

The main data structure maintained by the middlebox is the Connection hash table, which tracks the state of all established connections (both directions). Entries in the table are identified by the hash of *FlowId*, which consists of the source IP address, destination IP address, source port, and destination port. Each entry includes the following:

- *Flow definition*: source IP, destination IP, source port, destination port. [4 bytes per IP, 2 bytes per port; 12 bytes total]
- *Offset*: The difference between sequence numbers on the middlebox-source connection (generated as SYN cookies by the middlebox) and the destination-middlebox connection (chosen by the destination when a connection is established by the middlebox on behalf of a verified source). This offset is used to translate sequence numbers when the two connections are "spliced" at the middlebox. [4 bytes]
- *Timestamp*: Last time a packet was seen on this connection. Used to time out passive connections. [4 bytes]
- *Service bits*: (i) *pre-existing*: is this a pre-existing or spliced connection? (ii) *splice*: is sequence number translation required? (iii) *conformance*: has the source complied with traffic management measures (*e.g.*, responded properly to congestion control messages)?
- *InboundPacketRate*: Array of size $\frac{T}{t_i}$, containing the number of inbound packets seen for each interval of length t_i (T is the monitoring period). Used to mitigate unspoofed data flood attacks.

Preventing resource exhaustion and resolving collisions. To prevent the attacker from filling the Connection table with a large number of connection entries that have the same (unspoofed) source and destination IP addresses, but different port numbers, the middlebox maintains a separate Src-Dest table. This is a hash table indexed by the hash of the source IP - destination IP pair. For each pair, it keeps the count of currently open connections. Once the threshold is exceeded, no new connections are established. The value of the threshold is a system parameter, and can be changed adaptively depending on how full the Connection table is.

To resolve hash collisions between different flows in the Connection table, we use a Bloom filter-like technique and apply several hash functions until a vacant entry is found. If no vacancy can be found, the decision whether to drop the new flow or evict the old flow depends on the status of the latter and specific policy (see section 3.2).

The middlebox also maintains a secret key which is used as input to a hash function for generating unforgeable sequence numbers. This key is the same for all connections. It is re-generated at periodic intervals.

Handling connections originated from a protected network. In addition to keeping information about connections whose destination is the protected server, the middlebox also needs to keep information about the connections originating *from* the server in order to filter out packets with spoofed server address. This is done using a sliding-window counting Bloom filter [5]. (In our current implementation, we use filters with 3 hash functions.) During each time slice t_i, when a connection request from the server is observed, the middlebox adds connection parameters to the current Bloom filter B_i. If connection is terminated, it is removed from B_i.

3.2 Mitigation Policies

A large number of techniques for mitigating various types of DoS attacks have been proposed in the research literature. Virtually none have been deployed widely, due mainly to the lack of *transparency* and *scalability*: networking software must be modified at millions of end hosts, and performance penalty must be paid even when the hosts are *not* being attacked. Moreover, different attack types require different defenses, and supporting all of them (SYN cookies, capabilities, client puzzles, and so on) in a general-purpose TCP implementation is neither feasible, nor desirable.

Our main technical contribution in this part of the paper is to show how many anti-DoS defenses can be effectively and transparently implemented in the middle of the network at a minimal performance cost.

3.2.1 Mitigating spoofed attacks

The distinguishing characteristic of spoofing attacks is that the source addresses of attack packets are fake. For example, SYN flood is a classic denial of service attack, in which the attacker sends a large number of SYN requests to a TCP server. The server creates a half-open connection in response to each request. Once the server's queue fills up, all connection attempts are denied. In a spoofed data flood, the attacker simply floods last-mile bandwidth with spurious traffic. In Smurf-type and reflector attacks, the attacker sends packets with the victim's address in the source field to a large number of hosts, who then all respond to the victim, overwhelming him with traffic.

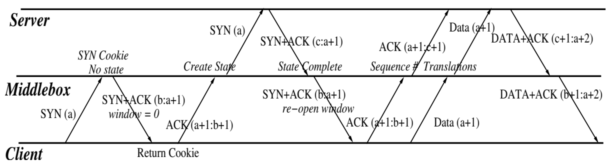


Figure 4: Outsourced SYN cookies with sequence number translation

Network-based SYN cookie generation. Our policy for mitigating spoofed SYN floods is shown in fig. 4. It is based on the well-known idea of *SYN cookies* [4, 20], except that, unlike the traditional approach, we do not require *any* modifications to the server TCP software.

After a dFence middlebox M has been dynamically introduced into all routes to some host S that is experiencing a denial of service attack, all traffic to S passes through M. On receipt of a SYN packet whose destination is S, the middlebox computes the SYN cookie as a cryptographic hash of connection parameters and the (frequently re-generated) local secret, adds the value of the cookie to the sequence number in the SYN packet, and uses it as the sequence number in its SYN-ACK response. No state is established at the middlebox at this stage.

Note that in the SYN-ACK response, *the middlebox M sets the receiver window size to zero*. Upon receiving the SYN-ACK with zero window size, C sends back an ACK packet and then enters the TCP Persist Mode. While in this state, C is not allowed to send any data packets with non-zero payload. So M effectively "chokes" C. M can "unchoke" C later by sending it any packet with a non-zero window size. If M receives data packets from C before the handshake with S is complete, it marks C as non-conforming and simply drops all further packets from C.

Note that, for a legitimate client that does not have correct implementation of persist mode, the middlebox will classify it as non-conforming and drop its packets.

It is important to prevent C from generating data packets before M completes its handshake with S (which may be far away from M). Otherwise, M has to buffer all these packets, which can be expensive. Dropping these data packets is not a good option because when the first few data packets at the beginning of a TCP connection are dropped, C can recover only through the TCP timeout mechanism. The default TCP timeout value is often set to 3 seconds, which can seriously degrade network performance as perceived by end users. We have confirmed this experimentally by turning off the choking mechanism, and observed a 3-second timeout each time. For server-side SYN cookies, choking is not needed because S will only receive packets from M *after* M has completed the handshake with a legitimate client, and thus all of S's packets can be safely forwarded to that client.

On receipt of an ACK packet from some client C, the middlebox M recomputes the cookie-based sequence number and verifies that it is correct. If so, the connection is not spoofed, and M creates new entries for it in the Connection and Src-Dest tables (see section 3.2). The entries are indexed by the hash of connection parameters. If a collision occurs, it is resolved as described in section 3.2.

At this point, M needs to establish a connection with the protected server S on behalf of the (verified) client C. This is done by performing the standard TCP handshake with S. M uses the same sequence number that C used in its SYN packet, by subtracting 1 from the sequence number in the ACK packet.

When M receives SYN-ACK from S, M forwards it to C and re-opens the window. There is a technical challenge here, however: the sequence numbers chosen by S for the $M - S$ connection are not the same as the cookie-based sequence numbers generated by M for the $C - M$ connection. As described in section 3.2, for every connection M maintains the offset between the two sequence numbers. On receiving SYN-ACK, C assumes that its previous ACK packet was lost and thus retransmits its ACK. C also exits the persistent mode as the SYN-ACK packet now has a non-zero receiver window size. M forwards ACK with proper sequence and acknowledgement numbers, thereby completing the handshake with S.

All subsequent data packets undergo sequence/ack number translation at M. When a packet arrives from S, M adds the offset to the sequence number. When a packet arrives from C, M subtracts the offset from the acknowledgement number. The *splice* bit is set in the Connection table to indicate that sequence number translation is required.

Spoofed data floods and reflector attacks. As described in section 3, the middlebox maintains information about client-originated connections (in the Connection table) as well as the connections originating from the server that is being protected (in the Bloom filter). Any data packet whose flow identification is not found in either of these two data structures is dropped. The same defense works against reflector attacks, because the middlebox filters out data packets from reflectors whose connection parameters do not belong to either the Bloom filter or Connection table.

3.2.2 Mitigating unspoofed attacks

Unspoofed data floods. The attacker can launch a data flood from a legitimate address by completing the TCP handshake and then flooding the bandwidth with data traffic. Our defense is based on enforcing compliance with congestion control measures.

When traffic rate on a connection exceeds some threshold value h (h is a tunable system parameter), the middlebox modifies ACK packets arriving from the server to reduce the receiver advertisement window, and starts measuring the rate of packets arriving from the client. (Recall from section 3 that the Connection table includes arrays for measuring packet rates on each unspoofed connection.) If at the end of the monitoring period the client's packet rate did *not* show a decreasing trend, the *conformance* bit is set to 0. All data packets on connections where the measurement period ended and the *conformance* bit is 0 are dropped. Note that this defense is feasible because a dFence middlebox controls *both* directions of TCP connections. The threshold h can be adaptively set based on the middlebox load.

Too many unspoofed connections. Many denial of service attacks involve the attacker opening a large number of connections from a legitimate IP address that belongs to a compromised, remotely controlled "zombie" machine. The zombie completes the TCP handshake, conforms to congestion control measures, and then overwhelms the server with a large number of requests. The Src-Dest table (see section 3) defends against multiple connections from the same address by limiting each source-destination pair to a reasonable number of connections.

NAPTHA attacks. In the NAPTHA attack, the attacker opens a legitimate connection, immediately closes it without sending FIN/RST, and opens another connection from a different zombie machine. This fills up the server's state, causing denial of service to legitimate connections.

To defend against NAPTHA attacks, the dFence middlebox maintains a timestamp for each connection, indicating the last time a packet was observed. If the idle time of a connection (the time since the last packet was observed) exceeds a threshold value (which is a tunable system parameter), the middlebox "times out" the connection by sending RST to the server. This is also done when the Connection table fills up, leading to a large number of collisions. The thresholds can also be determined empirically by analyzing attack traces.

Botnet attacks. dFence middleboxes can also support more sophisticated filtering policies. As an illustration, we sketch how they can be used to defend against botnet attacks. Our goal in this section is to demonstrate the expressiveness of dFence policies rather than describe a comprehensive solution to the botnet problem.

In botnet attacks, the attacker commands a large number of compromised computers to bombard the victim with HTTP or other requests. From the victim's viewpoint, this situation is similar to a flash crowd, since it is difficult to tell whether an individual connection is malicious or benign.

Our dFence-based botnet mitigation policy is based on source-prefix whitelisting. This policy is invoked only after the client has complied with all other measures, including congestion control. It gives preference to traffic from hosts in the white list of N most common /24 prefixes for a given server or network. This list can be created from traffic statistics, or else ISP customers can pay to be added to it.

The reason this approach is effective against botnets is that zombie machines tend to be sparsely distributed, *i.e.*, the attacker is likely to control only a handful of zombies within each /24 prefix. This observation is confirmed by our analysis of botnet traces collected by [9]. In both traces, fewer than 10 machines from any single /24 prefix are used during the attack. In trace I, 99% of prefixes have no more than 2 zombies, and in trace II, 99% of prefixes have no more than 7. In trace I, only 3 out of 22203 observed prefixes have more than 20 zombies, and in trace II, 48 out of 64667. (Note that the middlebox eliminates all spoofed connections using the anti-spoofing defenses described above, and that each bot is restricted to a modest amount of traffic by congestion control and compliance checking measures.)

This approach can be easily combined with an adaptive form of CAPTCHA-based Kill-bots [16]. The middlebox can adaptively redirect HTTP traffic from outside the preferred prefixes to a CAPTCHA server. This can be viewed as *rationing*: some fraction of the flows in the Connection table are allocated to the top N privileged flows, with the remaining (or un-privileged) flows competing for the rest of the table entries.

3.3 Policy Decision Logic

Because the dFence middlebox is in the data path of all connections to and from the servers that are being protected, it is critical to ensure that per-packet processing complexity is low and can scale to high link speeds. In particular, we want to avoid blindly applying different mitigation policies one after another regardless of the packet type.

On receipt of a packet, the middlebox first classifies it using TCP flag types. Depending on which flag is set (SYN, SYN+ACK, FIN, RST, *etc.*), it is sent to the respective processing function. For a SYN packet from client during the bootstrap or active phases, a SYN cookie is generated and SYN-ACK sent back to the client. For SYNs from server, the Bloom filter is updated. For SYN-ACKs from the server during the bootstrap or active phases, the Connection table is updated with the right offset value (difference between the seq/ack numbers on the middlebox-source and middlebox-server connections). During the removal phase, SYNs and SYN-ACKs are simply forwarded without updating the data structures at the middlebox.

For a data packet, its 4-tuple flow ID (IP addresses and port numbers) is looked up and checked against the Connection table and the Bloom filter to verify that it belongs to an established connection. If in the Bloom filter, the packets are forwarded. If in the Connection table, the *pre-existing* bit is checked and splicing performed, if needed. During the bootstrap phase, packets whose flow ID does *not* belong to both the Bloom filter and the Connection table are forwarded and middlebox state updated. During the active phase, they are assumed to be spoofed and dropped. During the removal phase, they are simply forwarded without seq/ack number translation or book-keeping.

The policy decision tree is depicted in fig. 5. "Is Cookie Correct?" represents re-computing the SYN cookie and comparing it with the acknowledgement number in the client's ACK packet. "To Apply Penalty?" represents checking that the client and its prefix are not generating too much traffic. "Can Replace Current Entry?" represents resolving collisions in the hash table. If the current entry is known to be compliant (*i.e.*, its *conformance* bit is set), then the new entry is dropped. If conformance is still being measured, the new entry is dropped, too. Otherwise, the old entry is evicted and the new entry is inserted in its place.

In all cases, processing is limited to a few hash table lookups, and access to the packet is limited to the information in the header (IP addresses, port numbers, sequence numbers, packet type). Detailed performance evaluation can be found in section 5.

3.4 Evasions and Attacks on the Middlebox

In this section, we focus on the denial of service attacks against the middlebox itself, and on techniques that an attacker may use to evade our defenses.

Exhausting the connection state. To prevent the attacker from filling up the Connection table, we use the Src-Dest table to limit the number of connections from any single host. For protection from botnets, we use source-prefix whitelisting as described in section 3.2.2. In general, resource exhaustion is prevented because the middlebox keeps state only for unspoofed sources that have complied with traffic control measures (*i.e.*, whose network-level behavior is similar to legitimate sources).

Adaptive traffic variation. The attacker may employ an ON/OFF attack pattern. On attack detection, the middlebox is introduced on the data path. As soon as middlebox is introduced, the attacker stops sending attack traffic. All legitimate traffic goes via the middlebox and suffers minor degradation due to triangular routing. After some time interval (the attacker assumes that the middlebox is now removed from data path), he starts sending attack traffic again, and so on. To provide a partial defense against this attack, we avoid rapid introduction and removal of middleboxes. Once the middlebox is introduced, it remains in the data path for some period even after the attack subsided. The duration of this interval is randomized.

Werewolf attack. The attacker starts by behaving legitimately, gets established in the Connection table, complies with congestion control requests, and then starts bombarding the server with attack traffic. We deal with this attack by periodically re-measuring traffic sending rates and source-prefix whitelisting.

Multiple attacks. The attacker can try to overwhelm the dFence infrastructure by launching multiple attacks on

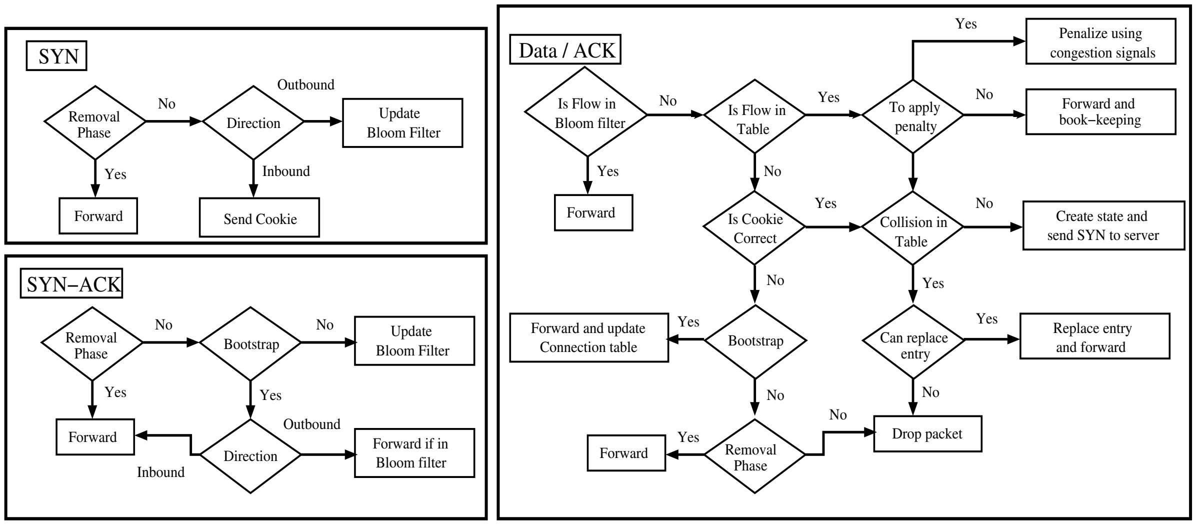


Figure 5: Policy decision tree

several destinations. We employ an adaptive provisioning strategy that scales up the number of middleboxes in the network with the number of attacked destinations (among those who have subscribed for dFence protection).

4 Implementation

Our prototype implementation consists of two components: (i) control-plane traffic interception, and (ii) data-plane attack mitigation. We prototyped the control plane functionality on a general-purpose processor using the extensible open-source router platform called XORP [13]. The anti-DoS data plane functionality is implemented on a special-purpose platform consisting of a Radisys ENP-2611 board, with a 600 MHz Intel® IXP2400 network processor.

The IXP2400 network processor contains one 32-bit XScale controller running Montavista Linux and eight 32-bit RISC cores called micro-engines (MEs). Each ME has a private 4K instruction store, onto which code for different packet processing functions (PPFs) is loaded. The micro-engines share a 16KB on-chip scratch-pad, off-chip SRAM (128MB), and DRAM (1GB).

The complete setup is depicted in Figure 6(a). The control plane uses BGP and IGP to make routing decisions and update the forwarding table. The data packets are handled on the fast path by IXP.

Control plane interception. The middlebox starts its operation after it receives the signal that a DoS attack has been detected. (Our focus in this paper is solely on mitigation rather than detection; dFence is compatible with any existing DoS detection mechanism—see section 6.) As discussed in Section 2.1, the middlebox intercepts traffic to the hosts experiencing the attack by sending iBGP advertisements to all routers within the same AS. Using BGP policy configuration in XORP, the local preference in the advertisements is set higher than the other routers. As a result, all border and intermediate routers make one of the middleboxes their next hop on the routes to the attacked hosts. Note that iBGP advertisements are sent only for the network prefix(es) under attack. To set up tunnels and ACL rules, the middlebox remotely configures the egress router. This is needed to prevent filtered packets from looping back to the middlebox—see Section 2.1.

Data plane mitigation. The attack mitigation policies are implemented on IXP network processors using the Shangri-La framework [19]. Shangri-La provides a flexible high-level programming environment that facilitates rapid development of packet-processing applications. We chose IXP over Click primarily for pragmatic reasons: the IXP multi-processor architecture supports multiple threads and hence provides higher throughput.

We implemented our mitigation policies as an application graph of packet processing functions (PPFs), operating on different packet types (SYN, data, and so on). The PPFs, as shown in Figure 6(b) are mapped to the IXP micro-engines using the Shangri-La run-time system.

Control-data planes interaction. The fast forwarding path on the data plane uses forwarding table entries established by the control plane to put the packets on appropriate output interfaces. We implemented the communication interface between the control plane (on XORP) and data plane (on IXP) using sockets and ioctl() system calls. Communication between a XORP process and a process running on the XScale processor occurs via standard C sockets, and communication between XScale and micro-engines occurs via ioctl() (see Figure 6(a)).

The XORP process sets up the MAC/IP addresses of the interfaces on the IXP data plane, and establishes the mapping between next-hop IP and port numbers. To set up the forwarding table, XORP runs BGP/IGP on control interfaces (on the host processor) and communicates the forwarding table entries to the IXP so that the data plane applications can use the table to forward data packets.

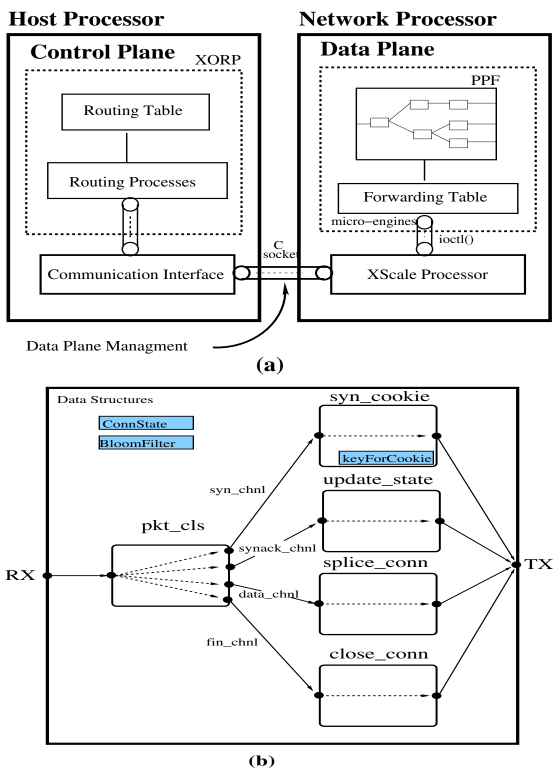


Figure 6: dFence System Implementation. (a) Control plane interception is implemented using XORP on a general-purpose processor. Data plane attack mitigation is implemented on Intel IXP network processors. (b) PPFs for attack mitigation policies.

5 Evaluation

In this section, we present an experimental evaluation of our prototype system. The IXP-based prototype implementation of the middlebox is described in section 4. Attack traffic comprising spoofed SYN packets, data packets, and spoofed ACK/RST/FIN is generated using IXIA packet traffic generator [15]. IXIA has 20 copper ports and two fiber ports. Each fiber port can generate up to 1490 Kilo packets per second, where packet size is 64 bytes.

5.1 Micro benchmarks

To measure throughput and latency of our attack mitigation policies, we directly connect the IXIA fibers ports to two optical ports on the IXP. Traffic generated using IXIA is processed by PPFs on the micro-engines. IXP 2400 has eight micro-engines, two of which are used for receive/transmit modules. We compose the application using four PPFs, each handling a particular packet type: SYN, SYN-ACK, data and FIN/RST. The four PPFs are mapped onto one micro-engine each. The PPF for packet classification is mapped to the same micro-engine as the PPF for FIN/RST. PPFs can be also be mapped to more than one micro-engine, where the code for the PPF is replicated on all the engines.

Packet Type	Packet Processing and Forwarding	Min	Max	Avg
SYN	SYN Cookie and SYN-ACK Generation	39.3	60.4	42.1
	Bloom filter update	25.28	27.06	25.86
	No processing	15.66	17.02	15.94
Inbound Data	Present in Bloom filter	23.24	24.8	23.56
	Present in Connection Table - splice	37.06	40.84	38.61
	Absent in both - forward (removal phase)	31.8	34.06	32.54
Outbound Data	Present in Bloom filter	23.24	25.3	23.56
	Present in Connection Table	37.66	41.64	39.1
	Absent in both - forward (removal phase)	29.52	44.92	30.15
	Absent in both - update (bootstrap phase)	31.5	33.6	32.1

Table 1: Latency benchmarks (in micro-seconds).

Synthetic traffic generated by IXIA consists of 100-byte packets. The maximum input traffic rate attainable in this case is 1041 Kilo packets per second. SYN packets are generated with arbitrary sequence numbers. Since our mitigation policies at the IXP drop packets with invalid sequence/ack numbers, we configure IXIA to automatically insert appropriate numbers into data and FIN packets. To enable testing over longer periods, we disable the interval-based key for generating SYN cookies. Instead, we use a single key that persists over the entire duration of testing using IXIA. This ensures that the data packets with appropriate seq/ack numbers (corresponding to those generated by the middlebox as part of SYN cookie generation) have their flow identifiers in the Connection table and are spliced properly by the IXP.

Latency. Table 1 shows the latency (in micro-seconds) introduced by the middlebox when dealing with different packet types and for different types of processing. Latency includes both processing and packet forwarding. Bloom filter update is performed only for SYN packets from the hosts that are being protected (all such connections are assumed to be legitimate). "Present in Bloom filter" checks the existence of flow ID (IP addresses and ports) in the Bloom filter, and forwards if present (*i.e.*, the packet belongs to an existing server-originated connection). "Present in Connection Table" checks whether the flow ID is present the Connection and, if so, forwards according to the status bits (splicing - seq/ack number translation; pre-existing - connection was classified as legitimate during the bootstrap phase). "Absent in both - forward" applies during the removal phase, when all data packets are simply forwarded. "Absent in both - update" applies during the bootstrap phase: middlebox state is updated for packets received from the protected server by setting the *pre-existing* status bit to *true*.

The latency of updating the Bloom filter (done only during bootstrap phase) is higher than checking the filter. For data packets, checking in the Connection table and splicing (seq/ack number translation + incremental TCP

Packet Type	Packet Processing and Forwarding	1 ME	2 ME	3 ME
SYN	SYN Cookie and SYN-ACK Generation	205	401	467
	Bloom filter update	530	1041	1041
	Forward	1041	1041	1041
Inbound Data	Present in Bloom filter	507	1011	1041
	Present in Connection Table - splice	264	525	781
	Absent in both - forward (removal phase)	326	652	974
Outbound Data	Present in Bloom filter	507	1011	1041
	Present in Connection Table	259	515	766
	Absent in both - forward (removal phase)	318	637	1029
	Absent in both - update (bootstrap phase)	326	653	951

Table 2: Throughput benchmarks in Kilo Packets Per Second (Kpps). Maximum input rate from IXIA (one fiber port) is 1041 Kpps with packet size = 100 bytes.

checksum computation) is more expensive than checking the Bloom filter, updating, or simple forwarding.

Throughput. Table 2 presents our throughput benchmarks. Throughput scales linearly as more micro-engines are allocated to the PPFs for all packet types and processing functionalities, except for SYN cookie generation. For the latter, maximum throughput supported by a single IXP is 467 Kpps.

5.2 End-to-end benchmarks

For end-to-end measurements, our server is a 1 GHz Intel P-III processor with 256 MB RAM, 256 KB cache, running an Apache Web Server on Linux 2.4.20 kernel. Legitimate traffic is generated using the httperf [23] tool which issues HTTP requests to the Web server. Both the client and the server are connected to a Gigabit Ethernet switch. Spoofed attack traffic is generated using IXIA, which is connected to the fiber optical port of the switch. All traffic goes via a routing element running XORP. For our prototype, we do not include attack detection and use a trigger to install the middlebox on the data path.

Our evaluation metrics are *connection time*, measured using httperf, and *max TCP throughput* attainable between a legitimate client and the server, measured using iperf [14].

Latency. In Figure 7(a), X axis represents the attack rate in Kilo packets per second (100-byte packets), Y-axis represents connection time in milliseconds. For server content, we used `ama` `m` homepage (copied on April 17, 2006). Its size is 166 KB.

For one legitimate connection, no attack traffic and no middlebox on the data path, connection time is 16.4 ms. With the middlebox on the data path, but still no attack, connection time increases to 16.5 ms. For ten concurrent connections, total connection time increases from 121.1 ms to 121.5 ms.

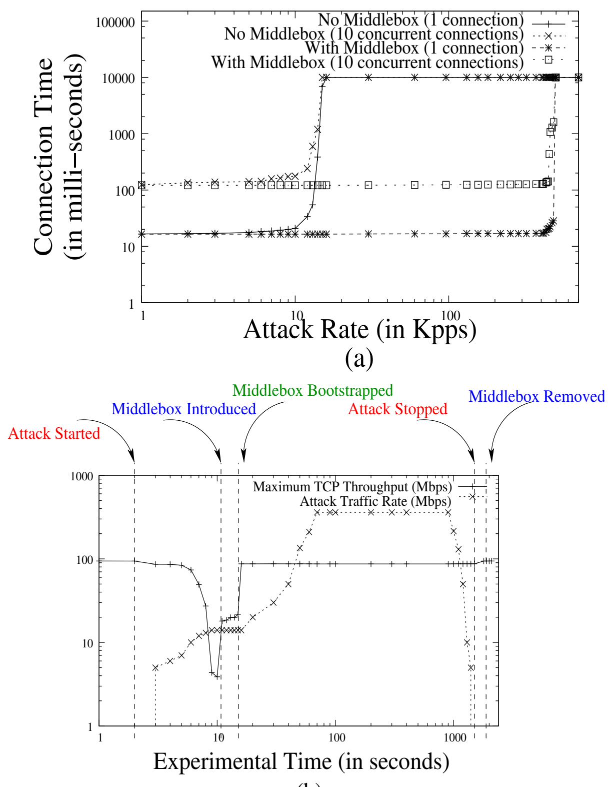


Figure 7: (a) End-to-end latency for one and ten concurrent HTTP connections to a Web server. Attack traffic rate is increased up to 490 Kpps (100-byte packets); (b) End-to-end maximum TCP throughput. Attack traffic rate, and TCP throughput are in Mbps.

As seen from Figure 7(a), connection time with no middlebox on the data path increases as attack traffic rate grows. After around 14 Kpps, the Web server can no longer handle the traffic and httperf client times out. The timeout interval is set to be 10 seconds. At this moment, the server dies. With the middlebox on the data path, connection time for legitimate clients remains constant even as attack rate increases all the way to 450 Kpps.

Throughput. Fig. 7(b) shows end-to-end performance (measured using iperf) over time as the server is attacked, middlebox enters dynamically into the data path, bootstraps, filters out attack traffic, and, after the attack subsides, is removed from the data path.

Before the attack starts, maximum TCP throughput between client and server is 94.3 Mbps. As the attack begins, it drops to 3.88 Mbps. After $t = 10$ seconds, the middlebox is dynamically introduced on the data path. During the 6-second bootstrap phase, the middlebox establishes state for ongoing connections, and throughput slowly increases to 21.7 Mbps (the increase is due to dropping of spoofed SYN requests - these packets do not get to the server, because the TCP handshake between the attacker and the middlebox is not completed). All data packets, whether spoofed or legitimate, are forwarded towards the server during the bootstrap phase (note, however, that the attack traffic rate stays below 14 Kpps). At $t = 16$, the middlebox enters its active mode, and starts aggressively profiling and filtering traffic. All spoofed traffic is dropped in this phase. Throughput now increased to 87.3 Mbps. At $t = 1500$, the attack stops, and the middlebox remains on the data path for the next 300 seconds. This interval (pre-determined) is used to time out the state for connections that were established via the middlebox during the active phase. At $t = 1800$, throughput returns to the normal (no attack, no middlebox) 94.3 Mbps level.

6 Related Work

Defenses against denial of service have been a subject of very active research, and the survey in this section is necessarily incomplete. Unlike previously proposed network-based defenses, dFence is completely transparent to the existing Internet infrastructure. Unlike proxy-based solutions, dFence uses novel dynamic introduction mechanisms to provide *on-demand* protection only when needed. dFence middleboxes can be quickly re-deployed to protect a different subset of end hosts without any modifications.

Network-based mitigation. Defenses based on secure overlays [2, 17] assume that all packets enter the network through the overlay's access points. The overlay checks each packet's legitimacy and filters out attack traffic. This method requires that the destinations' true IP addresses remain secret, and is thus difficult to combine with the existing Internet infrastructure. Similarly, Firebreak [11] assumes that the attacker does not know the targets' IP addresses, and that packets are tunnelled to the destinations by proxies deployed at edge routers. This requires software modification at legacy routers.

Defenses based on capabilities such as SIFF [36] and TVA [37] require that (i) destinations issue unforgeable tokens to legitimate sources, and (ii) routers filter out packets that do not carry these tokens. Both router and server software must be modified to support capabilities, and servers must be able to differentiate benign and malicious traffic. Flow Cookies [6] use the timestamp field in packets to insert cookies, and require server modifications to differentiate benign and malicious flows.

Pushback [21] rate-limits flows responsible for traffic congestion, and pushes filters upstream towards the sources of these flows. Router software must be modified. Rate-limiting is a coarse technique that does not differentiate between benign and malicious traffic, and may thus cause high collateral damage.

Cisco Guard [8] is a commercial product that dynamically redirects traffic to "cleaning centers" within the network. Traffic interception is not bi-directional; only traffic from client to server is intercepted. Cisco Guard applies several stateless filtering policies, and uses rate-limiting to reduce traffic volume (which may potentially cause high collateral damage). In contrast, our scheme intercepts both directions of traffic and supports both stateless and stateful policies to enable better differentiation between benign and malicious traffic.

Several designs for re-engineering the Internet infrastructure have resistance to denial of service attacks among their objectives [30, 31]. With indirection as the first-class principle of packet routing, these networks can easily reroute attack traffic to filtering devices by changing the mappings between identifiers and hosts. The scheme proposed in this paper is incomparable, because our goal is a solution that is fully compatible with and transparent to the existing Internet infrastructure.

Other network-based defenses, all requiring router modification, include route-based packet filtering [25], statistical analysis of incoming packets [18] and router throttles [38]. An evaluation of router-based defense systems can be found in [34].

Victim- and source-based mitigation. These defenses are deployed either at the servers, or at the ingress routers, and thus necessarily require substantial modifications to the existing software base. Server-based solutions also tend to be ineffective against last-mile bandwidth flooding attacks.

Kill-Bots [16] uses client legitimacy tests such as reverse Turing tests to differentiate between benign and malicious requests. In [32], victim servers encourage legitimate clients to "crowd out" malicious flows by sending higher volumes of traffic. In Pi [35], routers insert path identifiers into unused spaces within IP packet headers; servers then drop packets arriving on known attack paths. This requires modifications to both routers and servers, and may cause collateral damage if a legitimate source shares the route with an attacker. D-WARD [22] uses anomaly detection and compliance with traffic management measures to differentiate benign and malicious flows. Malicious flows are then blocked or rate-limited at source routers. Deployment requires wide-scale modification of router software. Ingress filtering [10] is limited to spoofing attacks, and also requires router modification.

Many methods have been proposed for *detecting* denial of service activity [12, 33, 28] and tracing back the sources of the attack [27, 29]. Our focus in this paper is on transparent, scalable *mitigation* rather than detection, and our solution is compatible with most proposed detection and traceback mechanisms.

7 Conclusions and Future Work

We described the design and prototype implementation of dFence, a novel network-based system for transparently mitigating denial of service attacks. The main advantages of the dFence middleboxes are their compatibility with the existing Internet infrastructure—they are introduced into the network using standard routing mechanisms, and their operation is completely transparent to the protected end hosts—and their ability to support a broad range of effective anti-DoS techniques. Control over both directions of TCP connections and efficient data structures for managing partial connection state enable several new defenses against denial of service, and make possible on-demand deployment of defenses in the middle of the network. Our experimental evaluation demonstrates that dFence provides effective protection against distributed DoS attacks at a minimal performance cost. Moreover, there is no impact whatsoever on traffic to servers that are not experiencing DoS attacks.

Future work includes investigation of mechanisms for configuration and management of dFence middleboxes, as well as design and implementation of an extensible scripting language for rapid development of new anti-DoS policies. Another research objective is a better understanding of *adaptive* attacker behavior and designing defenses against attackers who are aware of the anti-DoS middleboxes and deliberately craft their attack patterns to evade mitigation policies. This includes game-theoretic modeling of adversarial interaction between the middleboxes and the attackers. Finally, we would like to extend dFence to environments with multiple ISPs.

8 Acknowledgements

We are grateful to the anonymous NSDI reviewers and our shepherd Eugene Ng for insightful comments. Their suggestions have significantly improved our paper. We thank Tae Won Cho and Upendra Shevade for helpful discussions. Finally, we thank National Science Foundation for sponsoring the research under grant CNS-0546720.

References

[1] S. Agarwal, T. Dawson, and C. Tryfonas. DDoS mitigation via regional cleaning centers. Sprint ATL Research Report RR04-ATL-013177, January 2004.

[2] D. Andersen. Mayday: Distributed filtering for Internet services. In *Proc. USITS*, 2003.

[3] ANML. DDoS attack tools. h a m i h m , 2001.

[4] D. Bernstein. SYN cookies. h r ki h m , 1996.

[5] A. Broder and M. Mitzenmacher. Network applications of Bloom filters: A survey. *Internet Mathematics*, 1(4), 2004.

[6] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies along trust-boundaries (CAT): Accurate and deployable flood protection. In *Proc. SRUTI*, 2006.

[7] Cisco. Policy-based routing. h i m ar i h i h m, 1996.

[8] Cisco. Cisco Guard DDoS mitigation appliances. h i m r , 2007.

[9] D. Dagon, C. Zou, and W. Lee. Modeling botnet propagation using time zones. In *Proc. NDSS*, 2006.

[10] P. Ferguson and D. Senie. Network ingress filtering: Defeating denial of service attacks which employ IP source address spoofing. h a r r r h m , 2000.

[11] P. Francis. Firebreak: An IP perimeter defense architecture. h r ra i h ir r ak , 2004.

[12] T. Gil and M. Poletto. MULTOPS: A data-structure for bandwidth attack detection. In *Proc. USENIX Security*, 2001.

[13] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing extensible IP router software. In *Proc. NSDI*, 2005.

[14] Iperf. The TCP/UDP bandwidth measurement tool. h a a r r j r , 2003.

[15] IXIA. h i ia m m, 2006.

[16] S. Kandula, D. Katabi, M. Jacob, and A. Berger. Botz-4-Sale: Surviving organized DDoS attacks that mimic flash crowds. In *Proc. NSDI*, 2005.

[17] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *Proc. SIGCOMM*, 2002.

[18] Y. Kim, W. Lau, M. Chuah, and J. Chao. PacketScore: Statistics-based overload control against distributed denial-of-service attacks. In *Proc. INFOCOM*, 2004.

[19] R. Kokku, U. Shevade, N. Shah, A. Mahimkar, T. Cho, and H. Vin. Processor scheduler for multi-service routers. In *Proc. RTSS*, 2006.

[20] J. Lemon. Resisting SYN flood DoS attacks with a SYN cache. In *Proc. BSDCon*, 2002.

[21] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high bandwidth aggregates in the network. *CCR*, 32(3), 2002.

[22] J. Mirkovic, G. Prier, and P. Reiher. Attacking DDoS at the source. In *Proc. ICNP*, 2002.

[23] D. Mosberger and T. Jin. httperf: A tool for measuring web server performance. *Performance Evaluation Review*, 26(3), 1998.

[24] D. Pappalardo and E. Messmer. Extortion via DDoS on the rise. Network World, May 16 2005.

[25] K. Park and H. Lee. On the effectiveness of route-based packet filtering for distributed DoS attack prevention in power-law internets. In *Proc. SIGCOMM*, 2001.

[26] Prolexic. The Prolexic Zombie Report. h r i m r , 2007.

[27] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Network support for IP traceback. *IEEE/ACM Trans. Netw.*, 9(3), 2001.

[28] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang. LADS: Large-scale automated DDoS detection system. In *Proc. USENIX*, 2006.

[29] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-based IP traceback. In *Proc. SIGCOMM*, 2001.

[30] I. Stoica, D. Adkins, S. Zhuang, S. Shenker, and S. Surana. Internet indirection infrastructure. In *Proc. SIGCOMM*, 2002.

[31] M. Walfish, J. Stribling, M. Krohn, H. Balakrishnan, R. Morris, and S. Shenker. Middleboxes no longer considered harmful. In *Proc. OSDI*, 2004.

[32] M. Walfish, M. Vutukuru, H. Balakrishnan, D. Karger, and S. Shenker. DDoS defense by offense. In *Proc. SIGCOMM*, 2006.

[33] H. Wang, D. Zhang, and K. Shin. Detecting SYN flooding attacks. In *Proc. INFOCOM*, 2002.

[34] Y. Xu and R. Guerin. On the robustness of router-based denial-of-service (DoS) defense systems. *CCR*, 35(3), 2005.

[35] A. Yaar, A. Perrig, and D. Song. Pi: A path identification mechanism to defend against DDoS attacks. In *Proc. IEEE S&P*, 2003.

[36] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Proc. IEEE S&P*, 2004.

[37] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *Proc. SIGCOMM*, 2005.

[38] D. Yau, J. Lui, F. Liang, and Y. Yam. Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Netw.*, 13(1), 2005.

R-BGP: Staying Connected In a Connected World

Nate Kushman nkushman@mit.edu
Srikanth Kandula kandula@mit.edu
Dina Katabi dk@mit.edu
Bruce M. Maggs bmm@cs.cmu.edu

ABSTRACT

Many studies show that, when Internet links go up or down, the dynamics of BGP may cause several minutes of packet loss. The loss occurs even when multiple paths between the sender and receiver domains exist, and is unwarranted given the high connectivity of the Internet.

Our objective is to ensure that Internet domains stay connected as long as the underlying network is connected. Our solution, R-BGP works by pre-computing a few strategically chosen failover paths. R-BGP provably guarantees that a domain will not become disconnected from any destination as long as it will have a policy-compliant path to that destination after convergence. Surprisingly, this can be done using a few simple and practical modifications to BGP, and, like BGP, requires announcing only one path per neighbor. Simulations on the AS-level graph of the current Internet show that R-BGP reduces the number of domains that see transient disconnectivity resulting from a link failure from 22% for edge links and 14% for core links down to zero in both cases.

1 INTRODUCTION

It has long been known that during convergence, BGP, the Internet interdomain routing protocol, causes packet loss and transient disconnectivity. For example, Labovitz et al. show that a route change generates, on average, 30% packet loss for as long as two minutes [22]. Wang et al. report that a single routing event can produce hundreds of loss bursts, and some bursts may last for up to 20 seconds [35]. Both popular IP addresses with a lot of traffic as well as unpopular addresses suffer temporary disconnectivity because of BGP dynamics [25, 31]. Furthermore, BGP causes much of the lasting transient failures that affect Internet usability; our recent paper [20] shows that half of VoIP outages occur within 15 minutes of a BGP update.

BGP often loses connectivity even when the underlying network continuously has a path between the sender and the receiver. Indeed, in the above studies, the underlying network continuously has such a path. The Internet topology is known for its high redundancy, even when considering only policy compliant interdomain paths [13, 36]. Hence, transient disconnectivity due to protocol dynamics is unwarranted. The objective of this work is *to ensure that Internet domains are continuously connected as long as policy compliant paths exist in the underlying network*.

Past work in this area has focused purely on shrinking convergence times [9, 18, 29, 34]. Such approaches, however, are intrinsically limited by the size of the Internet and the complexity of the BGP protocol.

We take a fundamentally different approach. We focus on protecting data forwarding. Instead of trying to reduce the period of convergence, we isolate the data plane from any harmful effects that convergence might cause. Specifically, while waiting for BGP to converge to the preferred route, we set the data plane to forward packets on pre-computed failover paths. Thus, packet forwarding can continue unaffected throughout convergence.

Our failover design addresses two important challenges:

(a) Ensuring Low Overhead: The size and connectivity of the Internet make a naive advertisement of alternate failover paths unscalable. Announcing multiple paths to each neighbor could lead to explosion of the routing state, and announcing even a single failover path per neighbor could lead to excessive update traffic. Instead, in our design, a domain announces only one failover path to one strategic neighbor.

(b) Guaranteeing Continuous Connectivity: The real difficulty in using failover paths lies in ensuring connectivity while progressing from the failover state to the final converged state. Inconsistent state across Internet domains can cause forwarding loops, or lead domains to believe that no path to the destination exists even when such a path does exist. We address the consistency problem by annotating BGP updates with a small amount of information that prevents transient routing loops and ensures that forwarding is never updated based on inconsistent state.

Our solution, R-BGP (Resilient BGP), needs only a few simple and practical changes to current BGP. It precomputes a few strategically chosen failover paths and maintains enough state consistency across domains to ensure continuous path availability. R-BGP has these properties:

- Two domains are *provably* never disconnected by the dynamics of interdomain BGP, as long as the underlying network has a policy compliant path between them.
- Like BGP, R-BGP advertises only one path per neighbor, and thus the number of updates it produces is on a par with BGP.

We evaluate R-BGP using simulations over the actual Internet AS topology. Our empirical results show that, when a link fails, R-BGP reduces the number of domains temporarily disconnected by the dynamics of interdomain BGP from 22% for edge links and 14% for core links down to zero in both cases. Even in the worst case when multiple link failures affect both the primary and the corresponding failover path, R-BGP avoids 80% of the disconnectivity seen with BGP. Furthermore, R-BGP achieves this performance with message overhead comparable to BGP and reduced convergence times.

Ensuring that routing protocols recover from link failures with minimal losses is an important research problem with direct impact on application performance. Significant advances have been made in supporting sub-second recovery and guaranteed failover for intradomain routing in both IP and MPLS networks [11, 28, 32, 33], but none of these solutions apply to the interdomain problem. The main contribution of this paper is to provide immediate recovery guarantees for interdomain routing, using scalable, practical and provably correct mechanisms.

2 BGP Background

The Internet is composed of multiple networks, called domains or autonomous systems (ASes). ASes use the Border Gateway Protocol (BGP) to exchange interdomain information on how to reach a particular address prefix. A BGP update contains the full path to the destination expressed in terms of AS-hops. Each BGP router selects the best route for each destination prefix, called the *primary route*, and *advertises only its primary path* to its peer routers, sending them advertisements *only when routes change*.

BGP is a policy-based protocol. Rather than simply selecting the route with the shortest AS-path, routers use policies based on commercial incentives to select a route and to decide whether to propagate the selected route to their neighbors. The policies are usually guided by AS relationships, which are of two dominant types: customer-provider and peering [13]. In the former case, a customer pays its provider to connect to the Internet. In peering relationships, two ASes agree to exchange traffic on behalf of their respective customers free of charge. Most ASes follow two polices for routing traffic: “prefer customer” and “valley-free”. Under the “prefer customer” routing policy, an AS always prefers routes received from its customers to those received from its peers or providers. Under the “valley-free” routing policy, customers do not transit traffic from one provider to another, and peers do not transit traffic from one peer to another.

Finally, BGP comes in two flavors: routers in different ASes exchange routes over an eBGP session, whereas routers within the same AS use iBGP.

3 Related Work

Most prior work on improving BGP performance addresses control plane issues, such as reducing convergence time and the number of routing messages [9, 18, 29, 34]. A common tactic is to prevent paths that are not going to be useful from being explored during convergence. For example, BGP-RCN [30] augments a BGP update with the location that triggered the update, which, upon a withdrawal, saves BGP from exploring paths that also traverse the same troubled location, and hence are likely to be down. Another proposal [9] sends additional withdrawal messages to purge stale information from the network as quickly as possible. Our approach differs from the above prior work because instead of worrying about shrinking the convergence time, it protects data forwarding from the harmful effects of BGP convergence by providing failover paths.

Prior work on failover paths is mainly within the context of *intradomain* routing. For example, in MPLS networks, it is common to use MPLS fast re-route which routes around failed links using pre-computed MPLS tunnels [28]. In IP networks, there are a few proposals for achieving sub-second recovery when links within an AS fail [11, 23, 33], this work does not extend to the interdomain context because it assumes a monotonic routing metric, ignores AS policies, and requires strict timing constraints.

Lastly, prior work on failover paths in the interdomain context does not provide a general solution for continuous connectivity. The authors of [8] have proposed a technique for immediate BGP-recovery for dual-homed stub domains. Their solution, however, does not generalize to other types of domains, and requires out-of-band setup of many interdomain tunnels. Also, the authors of [36] propose a mechanism that allows neighboring ASes to negotiate multiple BGP routes, but in contrast to our work they do not use these routes to protect against transient disconnectivity or provide connectivity guarantees.

4 Why Not Reduce Convergence Time?

There are two broad ways to address packet loss caused by BGP convergence: limit how long BGP convergence lasts, or ensure that BGP convergence does not cause packet-loss. Much prior work has focused on the former approach, i.e., shrinking BGP’s convergence times [9, 18, 29]. We chose to explore the second approach for two reasons:

(a) Fast enough convergence is unlikely: Given the size of the Internet, it is difficult, if not impossible, to design an interdomain routing protocol that converges fast enough for real-time applications. The convergence time of BGP is limited both by the time it takes routers to process messages, and by rate-limiting timers instituted to reduce the number of update messages. There’s an inherent trade-off between these two however: set the timers too low and convergence is limited by the time it takes routers to process the additional updates; set the timers too high and convergence is limited by the time it takes for the timers to expire. Currently, routers use a timer called the Minimum Route Advertise Interval (MRAI) to limit the time between back-to-back messages to the same neighbor for the same destination to 30 seconds, by default. Griffin et. al. [14] show that we cannot expect a net gain in convergence time by reducing this default, yet real-time applications such as

VoIP and games cannot handle more than a couple seconds of disconnectivity, thus even a single MRAI of outage can be devastating for them [17].

(b) Focus on convergence limits innovation: Imposing strict timing constraints on convergence stifles innovations in interdomain routing. For example, it is desirable for interdomain routing to react to performance metrics by moving away from routes with bad performance [36]. However, such an adaptive protocol will likely spend longer time converging because it explores a larger space of paths and changes paths more often. A mechanism to protect the data-plane from loss during convergence will be a key component of any research into richer routing and traffic engineering options.

The rest of this paper details the problem and presents R-BGP (Resilient BGP), a few simple and practical modifications to BGP that ensure continuous AS-connectivity. We present R-BGP in the context of a single destination; this keeps the description simple but also complete, since BGP is a per-destination routing protocol. Further, we first describe the problem and solution at the AS level, referring to each AS as one entity and ignoring router-level details. Then in §7, we discuss router-level implementations.

5 Transient Disconnectivity Problem

Ideally, when a link fails, the routing protocol would immediately re-route traffic around it. But, in reality, BGP takes a long time to find another usable route, creating a *transient disconnectivity*, during which packets are dropped.

Consider the example in Fig. 1, where MIT buys service from both Sprint and a local provider Bob, who in turn buys service from AT&T and Joe. Traffic sent to MIT flows along the dashed arrows in the figure.

In BGP, an AS advertises only the path that the AS uses to reach the destination. Since all of Bob's neighbors are using his network to route to MIT, none of them announces a path to Bob, and Bob knows no alternate path to MIT. Thus, if the link between Bob and MIT fails, Bob can no longer forward packets to MIT and has to drop all packets, including those from AT&T and Joe. Eventually, Bob will withdraw his route to MIT, resulting in AT&T advertising the alternate path through Sprint. Now that Bob knows again a path to MIT, it resumes packet forwarding.

This is an example of a transient disconnectivity. Specifically, Bob has suffered temporary disconnectivity to MIT even when the underlying AS-graph contains an alternate path. Note that the harm of transient disconnectivity is not limited to the AS without a path. In this example, AT&T also suffers transient disconnectivity to MIT even though it knows of an alternate path.

In practice, transient disconnectivity is typical whenever routes change, and might last for a few minutes [22, 35]. This delay stems from several causes. First, a usable path might not be available at the immediate upstream AS

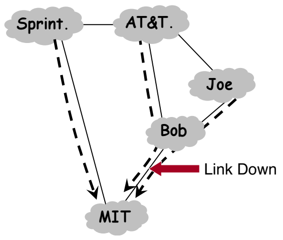


Figure 1: Transient Disconnectivity: When the link between Bob and MIT goes down, packets to MIT that are sent from Bob's domain and also those from upstream domains will be dropped until the alternate path through Sprint is advertised and BGP re-converges.

forcing the withdrawal to percolate through many ASes until reaching an AS that knows an alternate path. Further, searching for a usable path involves discarding many alternatives. For example, AT&T might first switch to the customer route "Joe→Bob→MIT" but will have to discard it when Joe reacts to Bob's withdrawal by withdrawing his route. Finally, this delay is exacerbated because a link failure creates a flurry of update messages for all destination prefixes that were using the link, thus delaying processing at nearby routers [7].

6 The Design of R-BGP

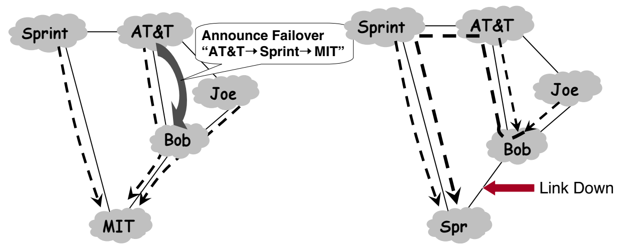
Our goal is ambitious; we want to ensure continuous connectivity between any two ASes as long as the underlying AS graph is connected. More specifically, our aim is for R-BGP to provide:

Continuous Connectivity: *If an AS has a policy compliant path both before and after a BGP routing event, then the AS should not become disconnected from the destination at any time during convergence due to the dynamics of interdomain BGP (eBGP).*

To achieve our goal, we use failover paths. Failover as an idea to solve the transient disconnectivity problem is conceptually simple; instead of searching for a new path to the destination after a link fails, as is the case in current BGP, pre-compute an alternate path *before* the link fails.

For example, in Fig. 1, with current BGP, Bob drops MIT's packets when its link to MIT fails. During BGP convergence, Bob learns of Sprint's path to MIT, uses it, and stops dropping packets. In contrast, in a failover solution, AT&T advertises to Bob the path "AT&T→Sprint→MIT", labeled as a failover path, as shown in Fig. 2a. As long as Bob's link to MIT is operational, the failover path is not used, and traffic follows standard BGP; but, if the link between Bob and MIT fails, Bob immediately diverts MIT's traffic to the failover path, i.e., he diverts the traffic to AT&T, who will forward it along the failover path to Sprint. As shown in Fig. 2b, the failover path saves Bob, Joe and AT&T from experiencing transient disconnectivity to MIT.

The conceptual simplicity of the failover idea hides three significant challenges.



(a) Announce Failover Path (b) Forward onto failover path

Figure 2: Failover Paths in Action: AT&T announces a failover path to Bob. When the link Bob→MIT goes down, Bob immediately forwards data onto the failover path ensuring all packets, even those from Joe and AT&T are not dropped.

- How to select and disseminate failover paths that ensure *continuous connectivity* without undue overhead?
- How to prevent inconsistent state across ASes from leading to transient loops during convergence?
- How to know that we have converged to the final state, i.e., we can stop using failover paths?

In the following three sections, we explain these challenges and the mechanism we use to address each of them.

6.1 Advertising Only a Few Failover Paths

The Internet is highly connected, with many alternate paths between a pair of ASes. Naively announcing failover paths can easily lead to an explosion of routing messages and state overhead in routers. How do we find a few strategic failover paths that achieve our goal of continuous connectivity regardless of which link fails?

6.1.1 To Whom Should Failover Paths Be Advertised?

In practice, a given AS is always incented to advertise a failover path to the neighboring domain through whom it is routing, because if this neighboring domain is left without an available path, it will drop the given AS's packets. ASes are less incented, however, to offer failover paths to other neighbors. In Fig. 2, AT&T has an incentive to advertise a failover path to Bob because if it does not, Bob may be left without a path and drop AT&T's packets to MIT. It is less incented, however, to offer a failover path to its competitor Sprint. Thus, with *R-BGP, an AS advertises at most one failover path per destination and only to the next-hop domain along its primary path.*

Advertising failover paths adds little update message overhead because each domain advertises at most one path to each neighbor, just like current BGP. To see this, recognize that an AS should not advertise its best path to the neighbor currently used to reach that destination, since this path would be a loopy (unusable) path to this neighbor. BGP's *poison-reverse* policy ensures that a withdrawal be sent in this case. R-BGP replaces this poison-reverse withdrawal with an advertisement of the failover path, keeping the overhead at a minimum.

The above rule also simplifies the implementation of failover paths in a way that is secure and has little forwarding overhead. If an AS were to announce multiple paths to a neighbor, it will need an additional signaling mechanism, such as marking the packets, to identify which path to use to forward packets coming from that neighbor. Such signaling mechanisms can be expensive, as the IP header has no free bits, and may require additional security mechanisms. Since R-BGP offers the failover path only to the neighbor used to reach the destination, only packets from this neighbor are forwarded on the failover path. This requires no additional signaling, and no additional security mechanisms to prevent abuse. Finally, though the failover path advertisement rule may appear too restrictive, we will prove that it is enough to achieve *continuous connectivity*.

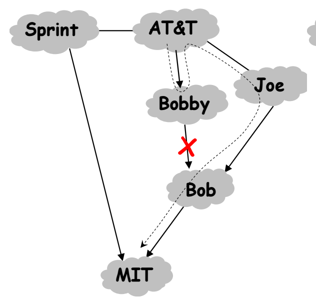
6.1.2 Which Failover Path to Advertise?

The above rule specifies the neighbor to which an AS advertises a failover path; it does not, however, answer the question–which failover path to choose in order to ensure continuous connectivity?

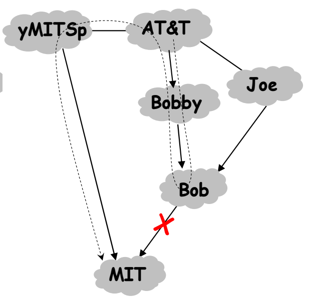
At first, it might seem that an AS should advertise the second-best route as a failover path, but it is likely that there is significant overlap between the primary and the second-best path, causing both paths to be unavailable at the same time. Consider the modified scenario in Fig. 3, where we inserted a new ISP, Bobby, between AT&T and Bob. Solid arrows represent primary paths to MIT and dotted lines represent failover paths. In this new scenario, AT&T's primary path goes through Bobby then Bob. Its second-best path is via Joe because ASes usually apply a "prefer customer" policy, and Joe is AT&T's customer whereas Sprint is AT&T's peer. Assume AT&T advertises its second-best path to Bobby, its next-hop AS on the primary, as a failover path. This failover path is useful if the link between Bobby and Bob fails, in which case Fig. 3a shows the path taken by the packets after the failure. The second-best path does not help, however, if the link between Bob and MIT fails. If AT&T had advertised the path via Sprint instead, it would have been possible to protect against either of the two failures. This leads us to the simple intuition – the more disjoint the failover path is from the primary, the more link failures it can protect against. Thus, our strategy for failover paths is:

Mechanism 1 - Failover Paths: *Advertise to the next-hop neighbor a failover path that, among the available paths, is the one most disjoint from the primary.*

We note a few important subtleties. First, a domain must check all paths it knows, *including failover paths*, to pick the one most disjoint from its primary. For example, in Fig. 3b, Bobby's most disjoint path is the failover path he learned from AT&T, and hence he advertises this path to Bob. Note that it may not be policy compliant to advertise this path to the next-hop neighbor. The failover path, however, will only be used for a short period during



(a) Second Best as Failover



(b) Most Disjoint as Failover

Figure 3: Which failover path to advertise? Choosing the path that is most link-disjoint from the primary path makes it less likely that a link failure will take down both the primary and the failover paths.

convergence, and it is used to guarantee connectivity to the advertising AS. Thus, we believe most ASes are willing to advertise such paths. Regardless, we show experimentally, that even using only policy compliant paths still eliminates most transient disconnectivity.

Second, note that the disjointness of two paths is defined in term of their shared suffix. In particular, any destination based routing protocol, including BGP, creates a routing tree to reach the destination. Once two paths going to the destination meet at an AS, they do not diverge again because no AS will announce multiple routes to the same destination. This means that at convergence two paths to the destination can only have a common suffix. The smaller the length of this common suffix, the less likely the two paths will fail simultaneously. Lastly, if multiple paths are equally disjoint from the primary path, then the normal BGP algorithm is used to choose between them.

6.1.3 Is This Enough?

The mechanism from the previous section maximizes connectivity while advertising only a few failover paths. But is this enough? Will every AS know a failover path for every link that can fail?

Let us go back again to the Bob-AT&T example introduced in Fig. 2 at the beginning of this section. When the link between Bob and MIT fails, Bob knows a failover path through AT&T. But Joe does not know any failover path; neither AT&T nor Bob sends through Joe, and thus none of them offers Joe a failover path.

We claim that it is not necessary for every AS to know a failover path for every link that can fail in order to achieve our goal. In fact, it suffices if *each AS is responsible only for the links immediately downstream of it*. The intuition for this is simple: if the AS immediately upstream of a failed link knows a failover path, packets of all upstream ASes are automatically protected. In the above example, as long as Bob knows a failover path for when the link Bob-MIT goes down, Joe's packets will see no loss. Further, in this example Joe is responsible for knowing a failover path only if the link between Joe and Bob fails, and AT&T is responsible for knowing a failover path only if the link between AT&T and Bob fails.

Thus, in order to achieve the first step in ensuring *continuous connectivity*, we need only show that Mechanism 1 ensures the AS immediately upstream of the down link always has a failover path on which to send packets. We show this by proving:

Lemma A.4. *If any AS using a down link will have a path to the destination after convergence, then R-BGP guarantees that an AS which is using the down link and adjacent to it knows a failover path when the link fails.*

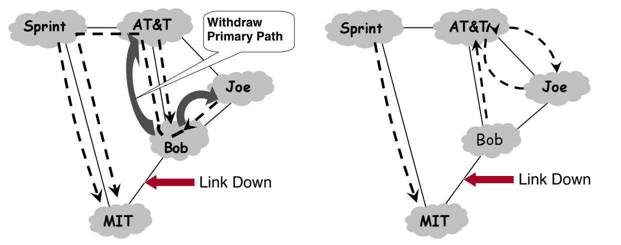
The formal proof is in the appendix, but the intuition is simple. Let Bob be the AS immediately upstream of the failing link. One of two cases will apply.

- *An AS upstream of Bob knows of a path p that does not use the failed link:* In this case, the most disjoint path at this upstream AS must not contain the down link because any path that contains the down link has a longer common suffix with the primary path through Bob and consequently is less link disjoint than *p*. As the failover path percolates from the upstream AS toward Bob, it can be replaced only with *more* disjoint paths, which necessarily do not contain the down link. Hence, Bob will be advertised a failover path that does not traverse the failed link.
- *No AS upstream of Bob knows of paths that do not use the failed link:* In this case, no AS using the down link knows an alternate path to the destination, and since ASes that do not use the down link will not change their advertisement as a result of the link going down, Bob and all other ASes using the down link will not have an alternate path, even after convergence.

6.2 Converging Without Routing Loops

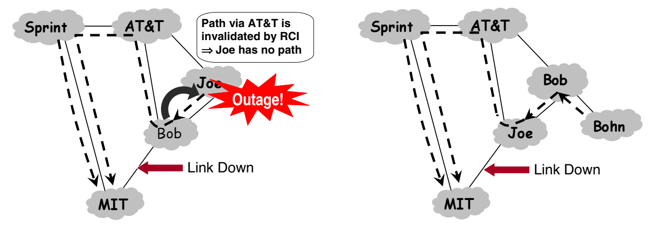
We have shown that announcing most disjoint failover paths guarantees that, whenever a link goes down, the AS immediately upstream of the down link knows a failover path and can avoid unnecessary packet drops. We now focus on the aftermath; specifically how to ensure that no AS *unnecessarily loses connectivity at an intermediate stage of convergence* either due to routing loops (§6.2) or because a usable path is no longer available (§6.3).

To see how a routing loop can be formed during convergence, we re-visit the previous example. In Fig. 4a, the link Bob→MIT is down and Bob has switched over to the failover path through AT&T. Now, Bob has no path to announce to AT&T or even to Joe because with normal BGP policies the path through one provider (AT&T here) is not advertised to another provider. Hence, Bob withdraws his route to MIT as in Fig. 4a. Unfortunately, these BGP updates do not indicate the reason for the withdrawal; so both AT&T and Joe believe that the other might be still



(a) Bob withdraws path (b) AT&T and Joe route via each other

Figure 4: Example of a Routing Loop



(a) Without Mechanism 3, Joe drops packets when Bob withdraws his path (b) Bob doesn't send John a withdrawal before he hears from AT&T

Figure 5: Avoiding disconnectivity while moving from a failover state to a converged state

be able to route through Bob, and thus together they mistakenly create a routing loop. In short, AT&T attempts to route along "Joe→Bob→MIT" while Joe attempts to route along "AT&T→Bob→MIT" causing the loop. Eventually, normal BGP will fix the loop but packets to MIT will be stuck in the loop and suffer drops until then.

We observe that the routing loop could be avoided if AT&T and Joe could determine that Bob's withdrawal renders the old paths through each other unavailable. This is possible if Bob includes in its update to Joe and AT&T *Root Cause Information (RCI)* indicating that the link between Bob and MIT is no longer available, preventing Joe and AT&T from attempting to route on any paths that use this link. This leads to our second mechanism:

Mechanism 2 - Root Cause Information: *Include in each update message Root Cause Information indicating which other paths will not be available as a result of the same root cause event.*

We defer the details of implementing RCI until §7.3. The idea of including in each update its root cause, however, is not new. Prior work [24, 30] uses root cause information to reduce BGP convergence time and number of messages. R-BGP benefits from the reduced convergence time and reduced number of messages provided by RCI, but is novel in using RCI to prevent routing loops during convergence. Assuming the valley-free and prefer-customer policies, we prove that using RCI eliminates transient routing loops:

Lemma A.9. *Consider a network that is in a converged state at time t, when a link fails, and converges again at $t + \tau$. At no time between t and $t + \tau$ do the forwarding tables contain any loops.*

The intuition underlying the proof is that loops occur because at least one AS tries to use an out-of-date route. RCI allows an AS to locally purge out-of-date routes, preventing the creation of transient loops. The details of the proof are in the Appendix.

6.3 Ensuring a Usable Path Throughout Convergence

We have just shown that if an AS has a usable path it will be loop free, but we have not yet shown that an AS will continue to have such a usable path throughout convergence. Continuing with our example in Fig. 5a, when Bob withdraws his path, Joe uses RCI to determine that the old path offered by AT&T is invalid. Joe concludes that he has no available path and starts to drop packets, even though AT&T will eventually advertise him a new path through Sprint. How do we prevent ASes, like Joe, from transiently losing their path? To solve this issue, we take the following approach:

Mechanism 3a - Use Old Primary Paths: *When left without a usable primary path, the AS immediately upstream of a down link forwards along the failover path, and all other ASes continue to forward along their old primary path.*

Thus, even though the path through Bob has been withdrawn, Joe can temporarily use the withdrawn path to forward packets to the destination.

Using old paths in this way, however, raises another question: How long can Joe use this old primary path? Ideally, Joe would *use the old primary path until one of his neighbors announces a new path or he knows that he will not have a path after convergence*. The first part is simple; If a neighbor announces a path, Joe just moves to the new path because, as we have just proven, RCI guarantees that this new path will be loop-free and will not traverse the down link.

The second part has a catch though– in current BGP an AS cannot tell whether a neighbor will eventually announce a path to him. This decision must be handled carefully, not waiting long enough may cause premature packet-loss, but waiting too long can create a deadlock state leaving an AS indefinitely forwarding along an old path. To precisely determine whether an AS will have a path after convergence requires an individual AS to ascertain a global property of the network using only the local information available to it. Griffin et al. have proven that even with access to the entire link state of the Internet, and all policies of all ASes, it is an NP-complete problem just to determine if the network will converge, without even attempting to determine the state to which it will converge [15].

The Internet has an inherent structure to it, however, and we take advantage of this structure to allow an AS to determine when it will know an available path. In particular, most paths on the Internet are valley-free [12], that is all ASes on the path pursue economic interests by offering the path only if it goes to or from a customer. Focusing on this common case of valley-free paths, allows R-BGP to provide the global guarantee of continuous connectivity, while using only local information available to each AS. We use the following Mechanism to communicate the required information to allow an AS to locally determine the global property of whether it will eventually have an available path:

Mechanism 3b - Ensuring Convergence: *An AS stops forwarding internally originated traffic along withdrawn primary paths or failover paths when explicit withdrawals have been received from all neighbors. An AS delays sending a withdrawal to a neighbor until it is sure it will not offer this neighbor a valley-free path at convergence.*

To see how this works, suppose another neighbor John is using Joe to get to MIT as shown in Fig. 5b. After Bob withdraws his route from Joe, Joe knows no route to MIT, until AT&T announces a new path. Joe waits until AT&T withdraws its current path, which contains the down link, or replaces it by a new path before sending a withdrawal to John, thus ensuring continuous connectivity for John. In contrast, it sends a withdrawal to AT&T as soon as it hears the withdrawal from Bob, since it knows it will not have a valley-free path to offer AT&T at convergence. This prevents a deadlock where Joe and AT&T are waiting on each other to send withdrawals.

More generally, an AS knows it will not offer a valley-free path to a non-customer once it has heard withdrawals or advertisements of non-valley free paths from all customers. Additionally, it knows it will not offer a valley-free path to a customer once it has heard withdrawals or non-valley-free advertisements from all neighbors. To identify paths which are valley-free, advertisements include an additional bit indicating whether or not a path is valley-free.

Since valley-free paths are also loop-free, enforcing that delayed withdrawals only follow valley-free paths allows R-BGP to ensure continuous connectivity in the common case when ASes are following valley-free and prefer-customer policies, and still avoid deadlock regardless of policies. Formally, Mechanism 3b ensures:

Theorem A.10. *Regardless of policies, in a converged state, no AS is deadlocked waiting to send an update, and no AS is forwarding packets along a withdrawn or failover path.*

Lastly, note that to ensure continuous connectivity, ASes continue to forward traffic received from their neighbors.

6.4 R-BGP: Intuition and Guarantees

Together, the preceding three mechanisms ensure continuous connectivity between two ASes as long as the underlying graph is connected. To understand this, recognize that when a link fails, the ASes most affected are those using a path through the down link. If we call the AS upstream of the down link the *failover AS*, then Mechanism 1 ensures that the failover AS will have a failover path when the link first goes down, allowing it to initially protect all traffic sent by the affected ASes. If we refer to alternate paths that do not contain the down link as *safe paths*, then Mechanism 2 ensures no loops throughout convergence by allowing an AS that changes its path as a result of the down link to avoid unsafe paths. Finally, Mechanism 3 ensures that, throughout convergence, all affected ASes protect their traffic by forwarding it on their old path to the failover AS, until they, or some AS between them and the failover AS, learn a safe path to the destination. Thus, the combination of all three mechanisms allows us to prove:

Theorem A.11. *Consider a network which is in a converged state at time t, when a link goes down, and converges again at $t+\tau$. Assuming the valley-free, and prefer-customer policies, if an AS A knows a path to destination p at times t and $t+\tau$, then at any time between t and $t+\tau$ the forwarding tables contain a path from A to p.*

The proof of this theorem is in the Appendix. This last theorem is the culmination of R-BGP design. It proves that R-BGP achieves the goal stated at the beginning of this section. Specifically, R-BGP ensures that any two domains stay connected as long as the underlying AS-graph has a valley-free policy compliant path that goes between them.

7 Implementation & Protocol Details

Our discussion so far considers each AS to be a single entity, but ASes are composed of many routers. Here, we describe a router level implementation of R-BGP; specifically, how R-BGP forwards packets, works with route reflectors, and communicates RCI. Fig. 12 in the appendix has pseudo-code for R-BGP's update algorithm.

7.1 Packet Forwarding

Packet forwarding with R-BGP is similar to normal forwarding except that when packets arrive at a given router, they may be traveling along either the primary path or the failover path, and the router will need to forward the packet differently in each case. This differentiated forwarding requires: a) detecting whether a packet is on the failover path or the primary, b) storing the next hop for both the primary and failover paths, and c) forwarding the packet to the appropriate next hop.

(a) Different Virtual Connections for Primary and Failover

To allow the router to distinguish whether the packet is traveling along the primary or the failover path, R-BGP utilizes two "virtual" layer-two connections between each pair of BGP-speaking routers, one for primary path traffic, and one for failover traffic. The simplest way to implement virtual connections is through virtual interfaces and virtual

LANs (VLANs). When the two routers are not physically connected, we use MPLS or IP tunnels. Thus, traffic sent out the primary virtual interface on one router, will arrive on the primary virtual interface on the other router, and similarly for the failover virtual interfaces. Supporting additional VLANs and virtual interfaces like this comes with little to no overhead, and is already heavily used both for configuring peering relationships between ISPs at public peering points and for configuring VPN customers [3].

(b) Storing Primary and Failover Forwarding Information

Routers can easily support the storage of separate forwarding entries for the primary and failover path through separate forwarding tables. Current routers already support separate forwarding tables for separate virtual interfaces [1]. While this simple model doubles the required forwarding memory, modern routers can accommodate such need. Since they need to support many VPNs, next-generation backbone routers are designed to handle a few million routes. In contrast, the largest Internet routing tables currently have only 200-300 thousand external entries [16], and this is expected to grow to only 370,000 in the next 5 years [26]. Any additional dollar cost associated with the added memory should not be prohibitive either because the forwarding memory typically represents less than 10% of the overall dollar cost of a router line card [1].

We can optimize the forwarding memory overhead by combining the primary and failover tables entries into a single integrated forwarding table. Most high speed router architectures have a level of indirection between the destination IP look-up, and the forwarding entries which contain the next-hop information used to forward the packet. The memory in the forwarding table is typically dominated by the IP look-up portion, because this is usually stored as a tree in a relatively memory inefficient way in order to facilitate fast look-ups. Note that R-BGP does not increase the number of entries (prefixes) in the forwarding table, rather it stores two pieces of information for each prefix – the primary and the failover next hop information. Hence, both primary and failover forwarding entries can be merged into a single table, eliminating the overhead of storing a second tree. The look-up tree need only be extended to store at each leaf two indices into the table storing next-hop information, one for the primary next-hop information, and one for the failover next-hop information.

Furthermore, it is possible to completely eliminate any need for additional memory on the line cards of the router. To do so, one stores the failover table on a specialized dummy line card in the router with no physical interfaces of its own. All packets arriving on any failover virtual interface on the router would be sent to this dummy line card, which would perform the look-up in its copy of the failover table, and ensure the packet is forwarded accordingly. One line card per router should provide sufficient capacity as long as multiple links connected to a given AS do not fail at the same time. This is similar to line cards built to handle tunnel encapsulation and decapsulation at line speed [2, 4].

(c) Forwarding Process

The path followed by a packet utilizing a failover path has at most three segments: (s1) the packet travels along a prefix of the old primary path along which packets were traveling before the link went down, (s2) the packet reaches the router immediately connected to the down link, who forwards the packet along its failover path, and (s3) the packet reaches a router that knows a primary path not containing the down link, and the packet is forwarded along that primary path.

When a router in (s1) receives a packet, it will receive the packet along a primary virtual interface. Since the primary virtual interface is associated with the primary forwarding table, the packet arrival will trigger a look-up in this forwarding table. For routers in (s1), the result of such a look-up will be the primary virtual interface towards the next-hop router on the primary path.

This continues until the packet reaches (s2), i.e., until it reaches the router immediately upstream of the down link. Since packets arrive at this router along a primary virtual interface they again trigger a look-up in the primary forwarding table. This router uses Bidirectional Forwarding Detection [19] to quickly detect the link failure and it populates all entries in its primary forwarding table that use the down link with the associated failover path entries instead. Thus, on this router, a look-up in the primary forwarding table will result in a failover virtual interface.

Thus, the packet will continue to be forwarded along failover virtual interfaces, with all look-ups performed on failover forwarding tables, until it reaches (s3), i.e., until it reaches a router that knows a primary path that does not contain the down link. This router will have an entry in its failover forwarding table that contains a primary virtual interface. From this point on to the destination, the packet will be forwarded along primary virtual interfaces, with look-ups performed only in primary forwarding tables.

7.2 Advertising Failover Paths with Route Reflectors

Large ASes utilize route reflectors to reduce overhead. Rather than have each router in the AS connect to every one of the BGP border routers (full-mesh), the route reflectors act as a central point where all externally learnt routes are stored and propagated to other routers. This works identically with failover paths; a border router that learns a failover path advertises the route to the route reflector, which in turn advertises its best failover path to the other routers.

7.3 Communicating Root Cause Information

BGP update messages are triggered by interdomain link state changes, intradomain link state changes, or configuration changes, which we call the root cause of the update event. Here, we focus on interdomain link state changes in the context of a single destination, but the same applies to other causes and destinations.

Root Cause Information (RCI) is created and forwarded whenever a link changes state. Note that when a link changes state, at most one of the the two routers adjacent to the link, i.e., whichever router uses the link for the given destination, will generate an update to its primary path. This *root cause router* attaches its AS number, a router identifier and the new link state to all updates generated as a result of the link state change. This information is called *Root Cause Information (RCI)*. Routers that change their routes as a result of receiving such an update message, copy the RCI in the received update into the update messages they generate. This allows any router whose paths are affected by this link state change to learn the unique root cause that triggered the series of update messages and purge *other* routes that include the down link from its routing table.

Encoding the appropriate link state change information is non-trivial, however, since updates triggered by different root cause events may propagate at different speeds. For example, if a link flaps (i.e., a link goes down and then immediately comes back up), the link down update may arrive at a router after an update from another neighbor announcing the link up event, causing the router to mistakenly assume the link is still down.

To solve this problem, R-BGP uses a monotonically increasing sequence number per BGP router and includes in the RCI both the identifier and the sequence number of the *root-cause router*. Further, in a router's BGP RIB, the AS-Path information includes for each AS in the AS-PATH, the *egress router* for that AS, and that router's sequence number. Upon receiving an update with RCI, the router purges a route if any of the route's AS-PATH entries matches both the AS and the router identifier in the update's RCI, and has a lower associated sequence number than the update's RCI. When an interdomain link goes down, all ASes on the affected AS-paths will receive updates with the root-cause router's RCI and will mark these paths as withdrawn. This is sufficient to prevent interdomain loops (Lemma A.9).

8 Experimental Evaluation of R-BGP

We evaluate R-BGP using simulations over the best known estimate of the Internet AS-graph.

8.1 Obtaining Internet Graphs and AS Policies

The most important test of an interdomain routing protocol is how well it performs in a full scale Internet setting. Thus, we evaluate R-BGP on a 24,142 node AS-level graph of the Internet generated from BGP updates at Route-views [6] vantage points. Additionally, we use the best known policy inference algorithms [10] to annotate inter-AS links as either customer-provider, provider-customer, or peer-peer (see §2).

The algorithms used to produce the policies have some limitations. First, the algorithms sometimes produce provider-customer loops. It is well known that such loops do not exist in the Internet, and could lead to persistent BGP loops [12]. Thus, we eliminate them by finding the AS in the loop with the fewest neighbors, and removing the edge between this AS and the next AS in the loop (its customer). Second, when one ISP spans multiple AS numbers, the algorithm in [10] assign these ASes a *sibling* relationship. We treat sibling relationships as peer-peer, since treating them as anything else also leads to provider-customer loops.

8.2 Simulator

BGP implementations in existing simulators such as SSFNET [5] and ns-2 [27] use so much memory that they cannot, even on our 8GB server, scale to the 24000-node AS graph of the Internet. To simulate the full Internet, we had to write our own BGP-specific simulator. Our simulator is optimized for performance, yet implements all the important characteristics of BGP convergence. In particular, it implements sending and receiving of update and withdrawal messages, detailed message timing including the MRAI timer, and the full BGP decision process including relationship-based route preferences.

8.3 Compared Protocols

We compare current BGP with three variants of R-BGP that differ only in the choice of the failover path.

- *Most-Disjoint Failover Path:* When ASes advertise the path most disjoint from their primary as a failover path, R-BGP guarantees that no unnecessary drops occur due to transient inter-AS disconnectivity. Though the most-disjoint path may not be policy compliant, ASes have an incentive to advertise it for failover. First, the cost is little–a failover path is only used for a short time until BGP converges. Second, as explained in §5, an AS that does not announce the most disjoint failover path risks having its own packets dropped upon a route change since a downstream AS may have no usable path.
- *Most Disjoint Policy Compliant Failover Path:* We also evaluate the performance of R-BGP when an AS advertises the most disjoint policy-compliant failover path. In particular, we would like to quantify how useful R-BGP would be if we limit failover paths to be policy compliant. If the likelihood of transient disconnectivity is very low, then it is probably sufficient to stick to policy compliant paths even for failover.
- *Second Most Preferred Failover Path:* It is also natural to ask what would happen if each AS advertises its second best path as its failover path, again limiting to only policy compliant paths. After all, when the primary path fails, an AS uses the second best path.

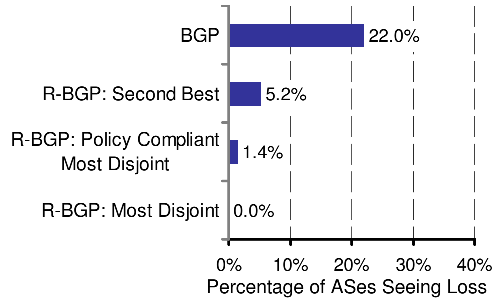


Figure 6: Singe Edge-Link Failure: Percentage of AS sources that will have a policy compliant path after convergence, but see transient disconnectivity to a dual-homed edge domain when one of its links fails.

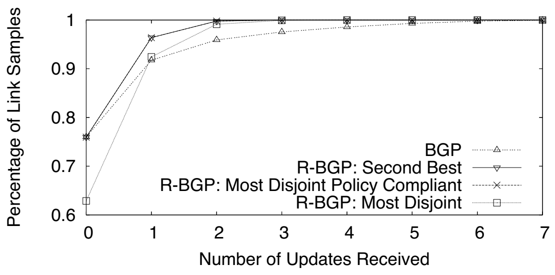


Figure 7: Number of Updates Per Link: CDF of the number of updates sent on each interdomain link. The measurements are from 18,400 simulations, where one of the two links of a dual-homed destination domain is taken down.

8.4 Experimental Setup

To ensure R-BGP works in all cases, we evaluate its performance for both edge link and core link failures.

(1) Dual-Homed Edge Domains: It is common in today's Internet for edge domains to be multi-homed. Multi-homing is sought after to improve resilience to access link failure. But how effective is such a backup approach in preventing transient disconnectivity? To answer this we look at the effect of taking down a link connected to a dual homed edge domain. For each of the 9200 dual-homed edge domains in our AS graph, we run a simulation, in which we take down one of the domain's two access links, and ask how many source ASes will experience transient disconnectivity to the domain. Specifically, we compare the performance of BGP and the R-BGP variants using the following metric: Among the sources that will be connected to the dual homed edge domain *after* BGP converges, what fraction will see disconnectivity during convergence? We also use this scenario to determine the performance of the various protocols when multiple links are taken down, and when a link comes back up at the same time that another goes down. Further, we quantify the relative overhead of the various versions of R-BGP by measuring the number of routing messages exchanged and the time to converge.

(2) Core Link Down: While the above scenario looks at access links, many more ASes can be affected when core links fail and so it's important to confirm that R-BGP avoids transient disconnectivity in these scenarios as well. We define a core link as a connection between two non-stub domains. In this scenario we compare the various protocols on the following metric: Among the AS pairs that were using the down core link, and will be connected *after* BGP converges, what fraction will see disconnectivity during convergence? For each of the 200 links that we tested, we ran 24142 simulations, one for each possible destination AS, and averaged the results across links.

9 EMPIRICAL RESULTS

Our empirical results confirm that in both the dual-homed edge domain scenario, and the down core link scenario, R-BGP prevents disconnectivity when a single link goes down. Additionally, even when multiple links go down simultaneously, R-BGP avoids almost 80% of the disconnectivity seen with BGP. Further, R-BGP achieves this performance with message overheads comparable to BGP and surprisingly improves convergence times.

9.1 Dual-Homed Domains Resilience to Link Failures

We first consider the dual-homed edge scenario. In particular, we explore the following question: How many source ASes are transiently disconnected when a dual-homed AS loses one of its access links? Fig. 6 reports the percentage of ASes that temporarily lose connectivity, averaged over each link for all 9200 dual-homed domains. The figure shows that, in BGP, 22% of the ASes experience transient disconnectivity when one of the links to a dual-homed destination domain fails. R-BGP using most disjoint failover paths reduces this number to zero, confirming the analytical guarantees.

The figure also shows that all variants of R-BGP significantly increase resilience to transient disconnectivity. R-BGP performs adequately when working within the confines of current policies; using policy compliant most disjoint paths for failover allows disconnectivity in only 1.4% of the cases. This means that even if ASes are not willing to temporarily provide transit for their non-customers, R-BGP still avoids almost all disconnectivity. Announcing the second best path as a failover path, though does not perform as well, allowing 5.2% of the domains to be temporarily disconnected. This is because often there is much overlap in the best and the second-best paths, reducing the number of link failures that the failover paths cover. Still, even an R-BGP variant that uses the second best path for failover is significantly better than current BGP.

9.2 Number of Updates and Convergence Time

Since R-BGP needs to maintain and update failover paths in addition to primary paths, one may be concerned that R-BGP may delay convergence or exchange a large number of update messages. Our results show that the number of update messages in R-BGP is comparable to BGP, and surprisingly R-BGP converges faster than BGP.

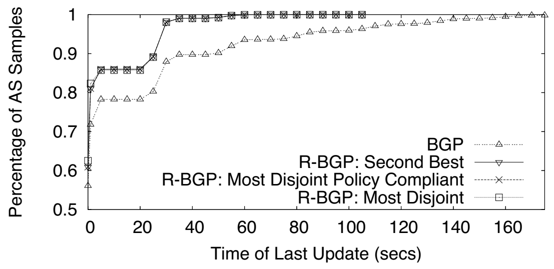


Figure 8: Convergence Time: CDF of the convergence time taken over AS samples. Convergence time is measured as the interval between the failure and the last time an ASes primary path forwarding table is updated. The measurements are from 18,400 simulations, where one of the two links of a dual-homed destination domain is taken down.

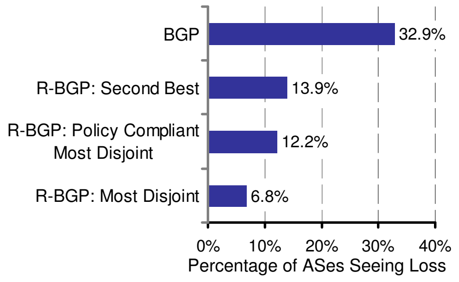


Figure 9: Simultaneous Link Failures: Percentage of ASes which have a policy compliant path after convergence, but see transient disconnectivity to a dual-homed edge destination, in the worst case scenario where one of its access links is taken down at the same time as a link on the failover path of the AS immediately upstream of the down link.

Again, considering the dual-homed edge domain scenario, Fig. 7 plots the cumulative distribution of the number of updates exchanged across each link in the graph, when routes converge after one of the two links of a dual-homed destination is brought down. The figure shows that 92% of interdomain links see at most one update during convergence for all protocols. R-BGP with most disjoint failover paths sends fewer messages than the other variants. Further, all variants of R-BGP send a number of messages comparable to BGP.

It might look surprising that R-BGP sometimes sends fewer updates than current BGP. This is due to the Root Cause Information (RCI) mechanism described in §6.2. In particular, when a link fails, current BGP may move to an alternate route that contains the same failed link and advertise this new route to neighboring ASes. RCI provides an AS with enough information to locally purge all routes that traverse the failed link, thus preventing such useless updates and significantly reducing path exploration. One may wonder whether RCI significantly decreases the number of messages and the failover advertisements eat most of this decrease, making the overall number of messages on par with BGP. This however is not the case. The number of messages with just RCI is only a bit smaller, but we omit these results here to enhance readability.

Fig 8 plots the cumulative distribution of the convergence time for each protocol when one link of a dual-homed domain is brought down. It shows that, on average, R-BGP tends to converge faster than BGP. Again this is because RCI eliminates unproductive path exploration during convergence. In our simulations, all R-BGP variants never took longer than 106s to converge, whereas BGP needed as long as 323s in certain cases. Again, RCI alone only does slightly better than R-BGP–it always converges within 96s, indicating that the overhead of the failover paths adds little to the convergence time.

9.3 Multiple Simultaneous Events

Our focus so far, including our analytical guarantees, has been on the case when only a single link goes down at a time. It is less likely for multiple link-down or link-up events to happen simultaneously especially at the interdomain level. Still, we show empirically that, even in this case, R-BGP can significantly reduce the chances of transient disconnectivity. To simplify interpretation, we again show this in the dual homed edge domain scenario.

9.3.1 Failure of Both Primary and Failover Paths

Rather than selecting the two failed links randomly, we simulate one of the worst cases for R-BGP, namely simultaneous link failures on both the primary and failover paths. As before, we pick the first link from among the two access links of a dual-homed edge destination. We pick the second link randomly from among the links on the failover path that would have been used to compensate for the failure of the first link. We perform a total of 9200 $\times$ 2 $\times$ 4 simulations, i.e., for each of the links of the dual-homed ASes, we randomly fail four different links on the failover path, one at a time.

Fig. 9 shows that R-BGP reduces the number of disconnected sources from 32.9% to 6.8%, avoiding 80% of the disconnectivity seen with BGP. The intuition is that even though the failover path of the first failed link is broken, the failover path of the second failed link may still be operational, thus avoiding disconnectivity.

9.3.2 Changing Failover Path During Failure

Can ongoing convergence on the failover path hinder the ability to ensure connectivity when the primary path fails? Our simulations show that R-BGP avoids transient disconnectivity and that no adverse interaction happens.

Specifically, we simulate the following worst case scenario for R-BGP. We pick one of the two access links of a dual-homed AS and bring down its preferred failover path by bringing down a link on that path. This triggers a change in the failover path. After the network has converged, we fail the access link of the dual-homed AS. At the same time, we bring up the previously failed link of most preferred failover path. This triggers a change in the failover path while it is in use.

Fig. 10 shows that despite ongoing convergence on the failover path, no packets are dropped when the primary

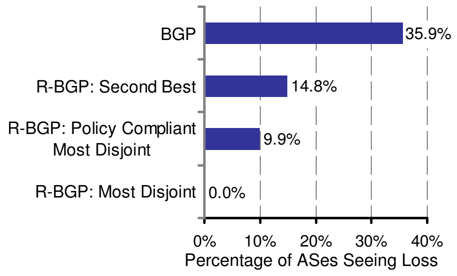


Figure 10: One Up One Down: Percentage of sources which have a policy compliant path with both links down, but see transient disconnectivity to a dual-homed edge destination, when one of it's links is taken down at the same time as bringing up a link on the default failover path of the AS immediately upstream of the first down link.

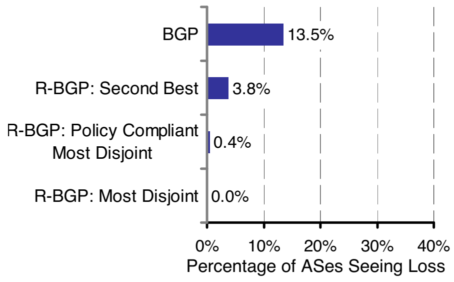


Figure 11: Single Core-Link Failure: When a random core-link is taken down, the fraction of source destination pairs using that link which see transient disconnectivity.

path goes down. In contrast, BGP causes over 35% of the ASes to lose connectivity to the dual-homed AS. Surprisingly, this number is significantly larger than the percentage of AS disconnectivity caused by BGP when a single link goes down. The fact that two events occurred together increases BGP transient disconnectivity despite that one of the events is a link up. This surprising fact is consistent with prior results that show that BGP might experience disconnectivity even when a single link comes up [35].

9.4 Resilience to Core Link Failures

Finally, we want to ensure that R-BGP performs well during core link failures in addition to access link failures. Thus, using the methodology described in §8.4, we test disconnectivity when a randomly chosen core link in the network fails. Figure 11 reveals that R-BGP using most disjoint failover paths eliminates packet loss for core link failures. Further, we see that core link failures cause proportionally less loss than edge links for all four protocols, because the highly connected nature of the core ensures that alternative paths are available to BGP. However note that even though a smaller fraction of ASes may see loss when a core link fails, failures on such links cause significantly more total packet loss as they carry much more traffic.

Additionally, note that R-BGP using most disjoint policy compliant paths sees the largest relative improvement, as we look at core link failures instead of edge link failures. To see why this is the case, note that if an AS remains disconnected after convergence, it cannot know a policy compliant failover path before the link fails but, if most disjoint paths, regardless of policy, are used the AS is guaranteed to have a failover path (Lemma A.4). Such persistently disconnected ASes and any AS that uses a path through such an AS will see transient disconnectivity when R-BGP restricts to most disjoint policy compliant failover paths. In our experiments, we see that persistently disconnected ASes tend to be long haul backbones like Abilene and GEANT that are primarily connected to stub ASes and don't have providers of their own. Such specialized long haul backbones are used more commonly by access networks that do not have a long haul backbone of their own in order to connect between regional offices. Since paths through a core link are less likely to involve this type of long haul backbone AS, a correspondingly smaller fraction of paths using a core link are affected by whether the most disjoint path is restricted to be policy compliant.

10 Conclusion

This paper shows that transient disconnectivity during BGP convergence is unwarranted and can be easily avoided. Our approach, called R-BGP, uses a small number of precomputed failover paths to protect data forwarding from the pathological effects of BGP dynamics. R-BGP performs a few modifications to current BGP that are easy to implement and deploy in today's routers. Simulations on the AS-level graph of the current Internet show that R-BGP reduces the number of domains that see transient disconnectivity resulting from a link failure from 22% for edge links and 14% for core links down to zero in both cases. Further, R-BGP achieves this loss reduction with message overheads close to current BGP, and, surprisingly, reduces convergence times.

11 Acknowledgments

We thank both Bruce Davie and Garry Epps of Cisco Systems, and John Scudder of Juniper Networks for answering our repeated questions on router implementation details. We thank Hariharan Rahul for extensive feedback on various versions of the paper. We thank our Shepherd Alex Snoeren for constructive comments on various versions of the paper. We also acknowledge the support of Cisco Systems and the National Science Foundation under NSF Career Award CNS-0448287. The opinions and findings in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or Cisco Systems.

References

[1] Private Discussions with Bruce Davie and Garry Epps of Cisco.
[2] Cisco GRE Tunnel Server Card. h i m i r r ar i imi r h m.
[3] Equinix Peering FAQ. h i i m ri a h m.
[4] Juniper Networks Tunnel Services PIC. h j i r r m .
[5] SSFNet. r .

[6] Univ. of Oregon Route Views. r i r .
[7] S. Agarwal, C.-N. Chuah, S. Bhattacharya, and C. Diot. Impact of BGP Dynamics on Router CPU Utilization. In *PAM*, 2004.
[8] O. Bonaventure, C. Filsfils, and P. Francois. Achieving sub-50ms Recovery upon BGP Peering Link Failures. In *Co-Next*, 2005.
[9] A. Bremler-Barr, Y. Afek, and S. Schwarz. Improved BGP Convergence via Ghost Flushing. In *INFOCOM*, 2003.
[10] X. Dimitropoulos, D. Krioukov, M. Fomendov, B. Huffaker, Y. Hyun, and kc claffy. As relationships: Inference and validation. *CCR*, 2007.
[11] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving Sub-Second IGP Convergence in Large IP Networks. *CCR*, 2005.
[12] L. Gao and J. Rexford. Stable Internet Routing Without Global Coordination. In *SIGMETRICS*, June 2000.
[13] L. Gao and F. Wang. The extent of AS Path Inflation by Routing Policies. In *Glob. Internet Symposium*, 2002.
[14] T. Griffin and B. Premore. An Experimental Analysis of BGP Convergence Time. In *ICNP*, 2001.
[15] T. G. Griffin and G. T. Wilfong. An analysis of BGP convergence properties. In *Proceedings of SIGCOMM*, pages 277–288, Cambridge, MA, August 1999.
[16] G. Huston. Growth of the bgp table - 1994 to present. h ar ,
[17] International Telecommunication Union. Connection Integrity Objective for International Telephone Service. E.855., 1988.
[18] L. Jiazeng et al. An Approach to Accelerate Convergence for Path Vector Protocol. In *Globecom*, 2002.
[19] D. Katz and D. Ward. Internet Draft. Bidirectional Forwarding Detection. draft-ietf-bfd-base-05.txt.
[20] N. Kushman, S. Kandula, and D. Katabi. Can you hear me now?! it must be bgp. In *CCR*, 2007.
[21] N. Kushman, S. Kandula, D. Katabi, and B. Maggs. R-BGP: Staying Connected In a Connected World. Technical Report TR-, MIT, 2007.
[22] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian. Delayed Internet Routing Convergence. In *SIGCOMM*, 2000.
[23] S. Lee et al. Proactive vs. Reactive Approaches to Failure Resilient Routing. In *INFOCOM*, 2004.
[24] J. Lou, J. Xie, R. Hao, and X. Li. An Approach To Accelerate Convergence for Path Vector Protocol. In *Globecom*, 2002.
[25] Z. Mao, R. Govindan, G. Varghese, and R. Katz. Route-flap Damping exacerbates Internet Routing Convergence. In *SIGCOMM*, 2002.
[26] D. Meyer, L. Zhang, and K. Fall. Internet Draft. Report from the IAB Workshop on Routing and Addressing, Dec 2006.
[27] The Network Simulator. i i am .
[28] E. Osborne and A. Simha. *Traffic Engineering with MPLS*. Cisco Press, 2002.
[29] D. Pei et al. Improving BGP Convergence Through Consistency Assertions. In *INFOCOM*, 2002.
[30] D. Pei et al. BGP-RCN: Improving BGP Convergence through Root Cause Notification. *Computer Networks Journal*, June 2005.
[31] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP Routing Stability of Popular Destinations. In *IMW*, 2002.
[32] E. Rosen, A. Viswanathan, and R. Callon. Multi-protocol Label Switching Architecture. RFC 3031.
[33] M. Shand and S. Bryant. IP Fast Reroute Framework, Oct 2005.
[34] L. Subramanian, M. Caesar, C. T. Ee, M. Handley, M. Mao, S. Shenker, and I. Stoica. HLP: A Next-generation Interdomain Routing Protocol. In *SIGCOMM*, 2005.
[35] F. Wang, Z. M. Mao, J. W. L. Gao, and R. Bush. A Measurement Study on the Impact of Routing Events on End-to-End Internet Path Performance. In *SIGCOMM*, 2006.
[36] W. Xu and J. Rexford. Multi-path Interdomain Routing. In *SIGCOMM*, 2006.

```
When an update is received:
a) Store AS-Path and Egress Router Sequence Number List in RIB-In
b) Mark withdrawn and out-of-date all RIB-In entries whose Egress
     Router Sequence Number List contains an entry with same AS
     and Egress Router as RCI, but with a lower sequence number
c) Compute new primary using the BGP decision algorithm over non-
     failover paths
d) Compute, over all paths, most disjoint path from the primary path,
     and set as the new failover path
e) If primary path changes and is non-empty, increment sequence
     number, send new primary path to all non-next-hop neighbors
     for whom it is policy compliant, and update primary forwarding
     table with new primary
f) If primary or failover changed and new primary path is non-empty
     send an update containing new failover, regardless of policy, to
     new primary next-hop neighbor, and update failover forwarding
     table with new failover
g) When no customer paths are marked out-of-date, if the primary
     path is not empty send withdrawals to all non-customers for
     whom the primary path is not policy compliant; otherwise the
     primary path is empty, so send withdrawals to all non-customers.
h) If new primary is empty path, no path is marked out-of-date path
     from any neighbor, no peer is offering a loopy path, and no
     provider is offering a loopy or backup path, then update for-
     warding table to stop forwarding locally originated packets, and
     send withdrawals to all neighbors to whom withdrawals have not
     already been sent, but continue forwarding along the old path
     packets received from neighboring ASes
```

Figure 12: Pseudo-code for the algorithm

APPENDIX

A Failover Proofs

An important step in our design is to formally prove that the proposed protocols achieve their goals.

We model the network as a directed graph in which each node represents a single-router autonomous system (AS). We analyze the scenario in which BGP is in a converged state at time t, then a link e goes down, and finally BGP converges again at time $t+\tau$. We assume that between times t and $t+\tau$ no other events occur, i.e., no other links come up or go down. We also assume that during this time no customer-provider-peer relationships between ASes change, and that no AS changes its policy regarding which routes are preferred and to whom those routes are advertised. Without loss of generality, we analyze the available paths to one prefix p during the transition. We use Q to denote the AS "upstream" of the failing link e, i.e., the AS which is forwarding packets for p along link e.

At various points in the analysis we make assumptions about routing policy. In order to state these assumptions, we must introduce several definitions. First, we assume that every edge in the network has a label indicating the relationship between its endpoints, either customer-to-provider, provider-to-customer, or peer-to-peer. Oppositely directed edges have reversed labels. We also assume that there are no cycles consisting entirely of customer-to-provider edges or provider-to-customer edges.

We say that a (directed) path in the network is *valley free* if a link labeled either provider-to-customer or peer-to-peer can only be followed by a link labeled provider-to-customer. We say that an AS A observes a *valley-free policy* if it never advertises a path from a non-customer to a non-customer.

Lemma A.1. *A valley-free path cannot contain a loop.*

Proof. See [21] for proof. □

We say that an AS A observes a *prefer-customer policy*, if A always prefers a path that is advertised by a customer of A over a path that is advertised to A by a provider or peer.

The last definition is more subtle. We say that an AS A follows a *widest-advertisement* policy if the following holds. For a prefix p, if A is ever willing to advertise a primary path to a neighbor B that it learned from another neighbor C, then whenever A knows of any path through C, it must advertise some path to B. Note that this latter path advertised to B need not have been learned from C. Further, note all three policies are consistent; ASes prefer customer paths, advertise the paths to the widest set of neighbors and their policies lead to valley-free paths.

Note that, our proofs hold for any route change for which policies incompliant with the assumptions are not exercised during convergence. With our terminology settled, we now prove several lemmas on our way to the theorems.

Lemma A.2. *If at time t the primary path from an AS A to p passes through e, but A knows of a primary or failover path that does not use e, then A's failover path to p does not use e.*

Proof. The proof follows from the fact that BGP establishes destination based routes. Thus, any two paths to p that share e must share a common suffix beginning with Q (the AS upstream of e). On the other hand, if a path does not contain e, then it cannot share this entire suffix. Hence this path must be more disjoint from A's primary path than any path that contains e. □

Lemma A.3. *If at time t there is any AS A whose primary path to destination p passes through link e, but who knows of a primary or failover path that does not use link e, then, at time t, Q, the AS upstream of e, knows of a failover path that does not use e.*

Proof. Let $U_0, \ldots, U_n$ denote the prefix of A's primary path to p, where $U_0 = A$, and $U_n = Q$. We prove by induction that for $0 \leq i < n$, U_i's failover path to p does not contain e, and hence U_i advertises a failover path that does not contain e to its successor, U_{i+1}, on the primary path. For the base case, by assumption and by Lemma A.2, U_0's failover path does not contain e. This failover path is advertised to its successor U_1 on U_0's primary path to p. For the inductive step, each U_i, $i > 1$, learns of a failover path that does not contain e from its predecessor U_{i-1}. Hence, U_i knows of at least one path that does not contain e, and by Lemma A.2, U_i's failover path, which U_i advertises to U_{i+1}, does not contain e. Finally, $Q = U_n$ learns of a failover path from its predecessor U_{n-1}. □

The following lemma assumes the widest advertisement policy and shows that if there is a path through Q from an AS V to destination p before e goes down, and V has a path to p after convergence, then Q knows a failover path when e goes down.

Lemma A.4. *If any AS using a down link will have a path to the destination after convergence, then R-BGP guarantees that an AS which is using the down link and adjacent to it knows a failover path when the link fails.*

Proof. First let us establish some notation. Suppose that some AS V uses a path through e to destination p at time t and knows of a path $P_0, P_1, \ldots, P_k$ to p at time $t + \tau$, where $P_0 = V$ and P_k hosts the prefix p. Note that $P_0, P_1, \ldots, P_k$ does not contain e, because e is down at $t+\tau$. Let i denote the largest index such that at time t the path from P_i to p passes through e. (Such an index must exist since the path from $V = P_0$ goes through e at time t.)

The key to the proof is to show that since each AS in the sequence $P_{i+1}, P_{i+2}, \ldots, P_k$ advertises a path to destination p to its predecessor at time $t+\tau$, it must also do so at time t. The proof is by induction on the path, starting at P_k and moving towards P_{i+1}. For the base case, we note that since P_k, which hosts p, knows a path to p at time t and advertises a path to P_{k-1} at time $t + \tau$, by the widest advertisement policy P_k must advertise a route to P_{k-1} at time t. By assumption, this path does not use e (e.g., it could be the one AS hop path to P_k). For the inductive case, P_j, where $i + 1 \leq j < k$ receives a path from P_{j+1} that does not use e, and since it advertises a path to P_{j-1} at time $t + \tau$, P_j must also advertise one at time t. As before, by assumption the path does not use e.

Now we apply Lemma A.3. Note that by the definition of i, P_{i+1} does not use a path through e at time t. Since, at time t, P_i uses a path through e and knows of a path from P_{i+1} that does not, by Lemma A.3, at time t AS Q knows of a failover path that does not use e. □

Lemma A.9. *Consider a network that is in a converged state at time t, when a link fails, and converges again at $t + \tau$. Assuming the valley-free, prefer-customer, and widest-advertisement policies, at no time between t and $t + \tau$ do the forwarding tables contain any loops.*

Proof. See [21] for proof. □

Theorem A.10. *Regardless of policies, in a converged state, no AS is deadlocked waiting to send an update, and no AS is forwarding packets along a withdrawn or failover path.*

Proof. See [21] for proof. □

As the the culmination of the preceding lemmas, the follow proves, assuming the widest advertisement policy, that R-BGP meets the continuous connectivity guarantee.

Theorem A.11. *Consider a network which is in a converged state at time t, when a link goes down, and converges again at $t + \tau$.Assuming the valley-free, and prefer-customer policies, if an AS A knows a path to destination p at times t and $t + \tau$, then at any time between t and $t + \tau$ the forwarding tables contain a path from A to p.*

Proof. By recognizing that ASes continue to indefinitely forward along their old path any packets they receive from neighboring ASes and invoking Theorem A which proves the forwarding tables cannot contain a loop, we can guarantee that no packet which is sent by A between t and $t + \tau$ will ever be dropped, and thus all packets must eventually reach the destination. Thus we most only prove that A will continue to forward it's own packets.

Recognize that by following Mechanism 3, A will continue to forward it's own packets unless it recieves a withdrawal from all neighbors. Thus we need only prove that A has at least one neighbor from whom it never recieves a withdrawal.

To do this, we prove by induction, that between t and $t + \tau$ A will never receive a withdrawal from N, the neighbor of A through whom A will forward at time $t + \tau$. We say the path that N offers to A at $t + \tau$ is $N_0, N_1, \ldots, N_k$, where $N_0 = N$ and N_k originates p.

As the base case it's clear that since N_k originates p and offers a path to N_{k-1} at $t + \tau$, it must also do so from t to $t + \tau$, by our widest advertisement policy assumption. Additionally, each AS, N_i, advertises the path from N_{i+1} to N_{i-1} at $t + \tau$ and by induction N_i is advertised a path by N_{i+1} from t to $t + \tau$. Thus, N must continuously advertise some path to N_{i-1} from t to $t + \tau$ by widest-advertisement policy. Analogously, N must offer A a path for the entire period between t and $t + \tau$. Thus we have shown that A will never receive a withdrawal from N between t and $t+\tau$, and so A will continue to forward it's own packets. □

Mutually Controlled Routing with Independent ISPs

Ratul Mahajan — *Microsoft Research*
David Wetherall — *University of Washington* and *Intel Research*
Thomas Anderson — *University of Washington*

Abstract – We present Wiser, an Internet routing protocol that enables ISPs to jointly control routing in a way that produces efficient end-to-end paths even when they act in their own interests. Wiser is a simple extension of BGP, uses only existing peering contracts for monetary exchange, and can be incrementally deployed. Each ISP selects paths in a way that presents a compromise between its own considerations and those of other ISPs. Done over many routes, this allows each ISP to improve its situation by its own optimization criteria compared to the use of BGP today. We evaluate Wiser using a router-level prototype and simulation on measured ISP topologies. We find that, unlike Internet routing today, Wiser consistently finds routes that are close in efficiency to that of global optimization for metrics such as path length. We further show that the overhead of Wiser is similar to that of BGP in terms of routing messages and computation.

1 Introduction

The Internet is made up of independent ISP networks that cooperate to carry traffic for each other and at the same time compete as business entities. This tension means that no one ISP is able to dictate traffic paths solely according to its best interests, but must instead acquiesce in some manner to the interests of other ISPs. BGP, as commonly used today, codifies one division of control: ISPs usually select the path of outgoing traffic and relinquish control over the path of incoming traffic. This is problematic because it means that ISPs may lack control where needed, e.g., to shift incoming traffic in response to a failure or temporary overload [4, 36]. Studies have also shown that routing paths, while often reasonable, can sometimes be poor from an end-to-end perspective [40, 44, 50].

This state of affairs is not new and has witnessed little real change over the past decade. Lacking effective protocol support, ISPs can mitigate problems through network engineering, e.g., by peering widely to minimize path length inflation in the common case [44], and by manually overriding configurations to handle very poor routes [27]. Newer routing platforms such as RCP [12] can also do a more effective job of optimizing routing within an ISP. However, while valuable, these approaches do not change the nature of the problems that exist.

Our intent is to develop an interdomain routing protocol that addresses these problems at a more basic level. We aim to allow all ISPs to exert control over all routes to as large a degree as possible, while still selecting end-to-end paths that are of high quality. This is a difficult problem and there are very few examples of effective mediation in networks, despite competitive interests having long been identified as an important factor [8].

While it is not a priori obvious that it is possible to succeed at this goal, our earlier work on Nexit [28] suggests that efficient paths are, in fact, a feasible outcome of routing across independent ISPs in realistic network settings. Specifically, Nexit shows it is possible for two ISPs to improve both their individual positions and end-to-end path quality by negotiating and making trades over their set of interconnection choices. So while the "price of anarchy" that measures the inherent inefficiency in multiparty models may be significant in theory [19, 39], Nexit suggests that it may be negligible for real networks.

However, Nexit is far from a complete or practical solution. It focuses on the limited problem of selecting interconnections between two ISPs, with no straightforward extension to routing across more ISPs. And it uses a negotiation mechanism that is much more heavyweight than BGP in terms of message and computational complexity. These factors limit its applicability in practice.

In this paper, we develop an interdomain routing protocol, called Wiser, that is complete and practical in the above senses of running across multiple ISPs and with overheads comparable to BGP. Wiser lets all ISPs exert a degree of control over all paths and produces high-quality paths. We undertake this design in a context that is compatible with independent ISPs: we do not require ISPs to disclose sensitive internal information (such as monetary transit costs) and we do allow ISPs to make decisions according to their best interests and their own optimization criteria (such as a mix of latency and utilization). Wiser paths are also completely policy compliant.

Wiser extends BGP with a simple coordination mechanism that builds on the bilateral ISP contracts that are already in place and is incrementally-deployable across pairs of ISPs. Each downstream ISP tags routing advertisements with costs that are similar to BGP MEDs. Each

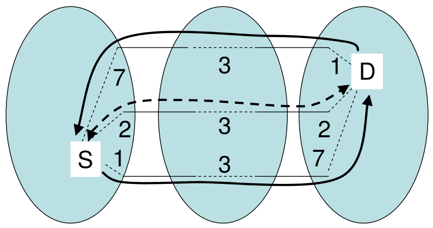


Figure 1: *Unilaterally controlled routing leads to longer paths and higher resource consumption inside ISPs. The solid lines depict early-exit routes. The dashed line depicts a route that is better overall as well as for the left and right ISPs.*

upstream ISP then selects paths with an amended decision process that considers the sum of its own costs and those reported by the downstream ISPs. This lets both upstream and downstream ISPs to exert control over all route choices. Wiser has built-in mechanisms to discourage potential abuse by ISPs. Prototypes in XORP [54] and SSFNet [46] show that its implementation complexity and message overhead are similar to BGP.

For ISPs to adopt this protocol, given no change in monetary compensation, it must be the case that all ISPs improve their position by following this protocol. We find this is so in nearly all cases because ISPs trade small concessions on some routes for larger gains on others.

In our experiments, the end-to-end paths with Wiser are comparable to the best that can be attained with a single entity selecting the entire path using complete information. Over measured ISP topologies, Wiser consistently comes close to the efficiency of globally optimized routing for several measures of interest to users and ISPs. With a latency metric, the most inflated 1% of paths are only 1.5 times longer than ideal, while they are 6 times longer with BGP defaults. With a bandwidth-sensitive metric, we find that Wiser reduces the ISP provisioning level needed to handle interconnection failures by an average of 8% relative to BGP.

These improvements in ISP and overall efficiency seem useful; however, the import of our work is that there exist alternative models for controlling routing among competing ISPs that are practical, policy-compliant, and provide high-quality paths. We consider this to be an issue that has been open since the original Detour study [40]. If we may generalize from our case study of Internet routing, a broader implication is that competing interests in a distributed system can be harnessed with practical protocols and in a way that they do not lead to poor efficiency.

2 Motivation

To see how unilateral control over routing paths can be problematic, consider Figure 1. The middle ISP interconnects to each of the other two ISPs at three locations. The numbers inside the ISPs represent the internal costs of carrying traffic along paths. We use rough path length as a visual surrogate for cost which will typically include latency and capacity considerations.

With typical business contracts, each ISP can select the route for its outgoing traffic. If each ISP acts individually, its best option is to choose the interconnection that minimizes internal cost. The result is the "early-exit" pattern, shown with solid lines, in which each ISP sends outgoing packets via the nearest interconnection. However, observe that both the left and right ISPs would be better off and end-to-end paths would be better if both ISPs were to use the middle exit. This is shown with the dashed line.

Unfortunately, there is no straightforward way to achieve this routing today. Path cost information can be exchanged between pairs of ISPs with MEDs. But MEDs are not transitive, and even so, their semantics simply change who controls routing; MEDs implement late-exit routing (in which downstream ISPs control the path) and neither provide joint control nor improve routes in the aggregate. Other available mechanisms such as communities have similar shortcomings.

Independent decisions imply that ISPs have poor control over their traffic. This is more pronounced for incoming traffic, for which the available methods are imprecise and coarse-grained at best [4, 36]. It also impacts outgoing traffic, for which ISPs today can only implement early- or late-exit routing but nothing in between. Our conversations with network operators confirm that this can be a major problem in practice.

Several studies have also observed inefficiencies as a result of unilateral control of routing [40, 44, 50]. They find many Internet paths to be slightly inflated and a small fraction to be highly inflated. The latter is most evident when paths traverse a chain of ISPs, since each ISP uses only a local view to select its portion of the route. These studies also find that a primary cause for inflation is how ISPs select their paths today, and other factors such as commercial preferences or inadequate peering are not major contributors [44, 50]. Although manual intervention can fix these routes, it is costly and error-prone. We interpret these studies as motivating a mechanism that consistently (and automatically) produces good paths.

As our example suggests, improving routing requires both sharing and incentives for overall decision making. The left ISP cannot know that the middle interconnection produces a better path without some sharing of information. Given only sharing, the left ISP has little incentive to act for the greater good and use the middle interconnection unless it is compensated, for instance, by the reverse traffic using the middle link as well. But by not reciprocating, the right ISP stands to decrease its costs at the expense of the left ISP. This can easily lead to a standoff. Our work aims to resolve such issues in a mutually-beneficial manner.

3 Problem and Approach

Our goal is to design a practical interdomain routing protocol that enables ISPs to jointly control routing and compute good end-to-end paths while acting in their own interests. In this section, we informally describe the constraints we place on the problem and our approach to it. We provide more detail in the next section.

For our solution to be practical, it must be incrementally deployable in the existing Internet. Pragmatically, this requires that it be comparable to BGP in terms of traditional overheads such as message complexity, computation, and convergence time. It also leads us to build on the existing business framework in which pairs of ISPs have peering contracts that only coarsely tie traffic levels to monetary compensation.

Handling independent ISPs raises a different set of issues [28]. We consider three constraints to be important:

1. Individual ISPs should improve their position by using our protocol compared to today. This is because we do not assume changes in monetary compensation; without compensation, ISPs will select routes that are to their own benefit. If a new protocol does not improve their position in the aggregate (by finding better paths within their networks), they simply will not run it. We refer to this property as *win-win*. While all more efficient routings will improve the overall situation, not all of them are win-win for each ISP involved, e.g., when one ISP is required to carry traffic further than necessary because it has the better network. Further, we want the win-win property to approximately hold even if one ISP abuses the protocol to try and take advantage of other ISPs.

2. While we need some information sharing, we should not require ISPs to disclose information in forms that they consider sensitive. This includes internal performance metrics and the monetary cost of carrying packets. Such information can be used against ISPs by competitors in the marketplace and is not disclosed today.

3. We should allow ISPs to set their own optimization criteria. In general, ISPs prefer paths that avoid congestion and minimize latency over some combination of their internal network and the performance experienced by their customers. However, different ISPs are bound to optimize paths differently, e.g., depending on their provisioning, monetary costs, the needs of their customers, and other factors of which we cannot be aware. There does not exist a universal metric that works for all ISPs, and past attempts to define such metrics have failed [18].

These constraints rule out known approaches to optimizing interdomain routing (§8). Our approach is to share rough path cost information between ISPs and enable each to improve its routing by its own reckoning. This does not explicitly improve end-to-end paths; rather, better overall paths are a welcome side-effect of better paths for individual ISPs. These paths do depend on the metric each ISP uses and so they cannot, for example, improve end-to-end latency if ISPs optimize strictly for utilization. But in practice, reasonable ISP metrics will factor in both delay and congestion. Our results then show that improving paths for ISPs is sufficient to avoid egregiously bad overall paths, bringing most of the benefit of a more direct but infeasible global optimization [20].

To share information, we extend BGP. ISPs tag advertised routes with costs that are derived from internal path costs. These advertised costs are *agnostic* in that they are simply cardinal numbers whose relative magnitude is significant. They serve to coordinate ISPs without requiring a standard metric or cost derivation methodology. We believe they limit the information that is shared to an acceptable level; they resemble MEDs – ISPs often use (cardinal) IGP costs to set MEDs [29, 30] even though MEDs have ordinal semantics – and not transparent measures that disclose information in defined units, e.g., latency in seconds or monetary cost in cents. It is of course possible that ISPs may attempt to reverse-engineer network properties from agnostic costs, but this is hardly a new capability because even today outsiders can measure ISP networks [45].

To enable ISPs to improve their position, we build on bilateral ISP contracts with a simple mechanism that coordinates the route selections of each ISP with those of its neighbors. An ISP selects paths based on the combination of its own costs and the costs advertised by its neighbor, and in return the neighbor does the same. Both track how costs are used to see that this is so. This mechanism is based on the observation that the interaction between ISPs spans many flows over time. It is not necessary that each ISP come out ahead for each individual flow. Rather, routing can be close to win-win when ISPs take a holistic view of traffic, compromising on some flows in exchange for bigger gains on others.

4 Design of Wiser

We now describe our routing protocol, beginning with the problem formulation.

4.1 ISP Model and Goal

Consider an internetwork of ISPs. Let each ISP associate a cost with each of their internal paths. We assume that each ISP aims to reduce the average cost of carrying traffic by its own measure, i.e., minimize the sum of the cost of its paths weighted by the traffic that they carry. So if $\mathsf{intcost}_I(p)$ is the internal cost of path p in ISP I, and $\mathsf{traffic}_I(p)$ is the rate of traffic carried along it over some period, we have:

$$\mathsf{cost}_I = \min_{paths} \left(\sum_{p \in paths} \mathsf{traffic}_I(p) \times \mathsf{intcost}_I(p) \right)$$

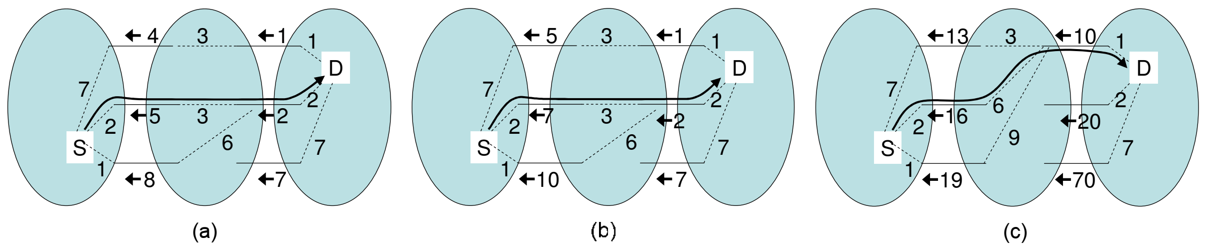


Figure 2: *(a) Traditional shortest path routing with comparable costs. (b)* Wiser *routing with agnostic costs approximates overall shortest path routing. (c) ISPs can unduly bias routing without cost normalization.*

This is close to what IGP protocols such as OSPF and IS-IS achieve today for an intradomain workload with internal costs that are the sum of link weights. The problem we tackle is that BGP does not achieve this goal for an interdomain workload due to the peering point selections that result from the interaction of independent ISPs.

We do not mandate how ISPs set internal costs, since we wish to let them use their preferred method. In our evaluation, we use IGP cost, i.e., the internal cost of a path is the sum of its component link weights. Further, we assume that ISPs act reasonably, for instance, by setting internal costs to favor paths with shorter internal distances and disfavor congestion. To reduce distance, they can simply set the link weights to reflect a measure of delay. To factor in utilization, they can use mappings that assign higher cost to links with a higher utilization [14]. Minimizing the sum of such costs then finds paths that disfavor congestion and are otherwise short.

We also need to measure the overall efficiency across ISPs so that we can assess the benefits of different schemes. If the internetwork were treated as one large ISP, with a single method of assigning costs, we could compute the routing with minimum global cost. However, there is no well-defined optimum if ISPs use different internal cost metrics. In such systems, the best solutions that can be obtained are *Pareto-optimal*. A solution is Pareto-optimal if the cost for any party cannot be reduced without hurting at least one other party. Pareto-optimality rules out solutions with obvious wastage when all parties could be better off. Unilateral routing is Pareto-optimal for individual paths (because each is best from the selector's angle) but not when considering all paths of an ISP in aggregate. We leave the goal of overall efficiency as an informal one. In our evaluation, we look at effects on overall measures of latency and bandwidth to see the impact of independent ISPs, and we compute efficiency only within individual ISPs.

4.2 Lowest-Cost Routing Across ISPs

Wiser adapts lowest-cost routing to the setting of independent ISPs. The above discussion suggests that, if we can put ISP interests aside, we can achieve efficient routing in a simple manner: have ISPs use the same method of assigning costs and run a traditional lowest-cost routing protocol over them. This is shown in Figure 2(a). Here, a route is computed to send traffic from source S to destination D. Each ISP has internal costs, which for ease of exposition we make static and symmetric in both directions of a link. Each ISP advertises to its neighbor the total cost to reach the destination, which is the sum of internal costs thus far. Each router selects the lowest cost path. The dark line shows the optimal route that is found.

However, a naïve lowest-cost routing protocol that does not handle ISP interests may fail to find such routes for several reasons. Costs may not be comparable and thus cannot simply be added across ISPs. Even so, optimal routes may require one ISP to lose for the greater good; we have argued that this is undesirable without changes in monetary compensation. Moreover, there is nothing to prevent ISPs from biasing their advertised costs to suit their own interests, inflating some and reducing others. Finally, even with reasonable cost advertisements, nothing prevents ISPs from ignoring others' costs while selecting paths.

The following subsections describe how we address these problems to approximate lowest-cost routing with independent ISPs. We use *cost normalization* to handle incomparable metrics, find win-win solutions, and incent ISPs to report costs honestly. We use bounds on the ratio of *usage cost* to incent ISPs to select paths honestly.

As a tool for autonomous parties, how Wiser will be used is likely to differ across ISPs. Our intent is to design it such that its works reasonably well by default for common situations and anticipated uses. Similarly, our mechanisms are not intended to completely prevent unwanted behavior. In fact, there is a fundamental conflict between efficiency and being provably cheat-proof [7, 33]. We favor efficiency because we expect honesty to be the common case. Competitors tend to act honestly when they seek mutual gains over a default contract [38], and even today ISPs cooperate using mechanism that are not cheat-proof. Cheating tends to be a poor strategy in long-term relationships such as those between ISPs because it risks long-term harm [1, 31].

4.3 Cost Normalization

In Wiser, ISPs normalize the costs received from neighbors so that they are comparable to internal costs. Each

ISP scales the costs it receives from a neighbor such that the sum of the costs received from the neighbor equals the sum of the costs advertised to the neighbor. This requires border routers to share information in order to determine the totals; we discuss how it may be implemented in §5. Once a normalization factor has been computed, ISPs can add their internal costs to normalized external costs to propagate routes. Routers then select the path with lowest total cost and advertise it upstream, as before.

If $\mathsf{advcost}_{I\to N}(P,d)$ is the cost advertised for destination d by ISP I to N over peering link P then we have:

$$\begin{aligned}
\mathsf{totaladv}_{I\to N} &= \sum_{P,d} \mathsf{advcost}_{I\to N}(P,d) \\
\mathsf{normal}_{N\to I} &= \frac{\mathsf{totaladv}_{I\to N}}{\mathsf{totaladv}_{N\to I}} \\
\mathsf{advcost}_{I\to N'}(P',d) &= \min_{P,N} \Big(\mathsf{intcost}_I\,(P'P) + \\
&\qquad \mathsf{normal}_{N\to I} \times \mathsf{advcost}_{N\to I}\,(P,d) \Big)
\end{aligned}$$

Above, $\mathsf{normal}_{N\to I}$ is the normalization factor that ISP I applies to the costs it receives from neighboring ISP N. The final equation is the propagation rule: each ISP advertises to its neighbors its best cost route: the minimum sum of its internal cost and the normalized cost it receives from its neighbors (that are most preferred).

Figure 2(b) shows how normalization approximates efficient overall routes. The total advertised cost from the right ISP to the middle ISP is 10. For simplicity, we have not shown the routing advertisements for S which travel from left to right. Assuming symmetry, the costs from the middle to the right ISP will be the same as those advertised from the middle to the left ISP, which have a total of 22. The normalization factor for costs coming in from the right ISP to the middle one is thus $\frac{22}{10}$. The top interconnection between these two ISPs, for example, is normalized to roughly 2 ($1 \times \frac{22}{10}$), and is propagated as 5 after adding 3 for the internal path costs. While the advertised costs are somewhat different than lowest-cost routing, the same globally shortest path is selected.

Normalization brings two key benefits as well as allowing costs to be compared across different ISPs. (In contrast, MEDs received from different ISPs are semantically incomparable, which can lead to instabilities [29] and other practical problems [47].) First, it limits biases that a dishonest ISP can cause by manipulating costs. Without normalization an ISP could trivially control routing by scaling its costs. For instance Figure 2(c) shows what happens if the right ISP inflates its costs by a factor of ten and the other ISPs continue to minimize the sum of costs. The new path is unfavorable to the middle ISP and has worse overall properties. Normalization nullifies the impact of such inflation.

The bias of relative changes in advertised costs is also limited because increasing the relative values of some costs implies decreasing the relative values of others. We find that even with complete knowledge, relative changes gain little for the downstream and inflict little damage on the upstream compared to not running Wiser (§6.3). And in the more realistic case of partial information, the outcome of manipulation can be hard to predict and possibly inferior for the downstream. For instance, consider Figure 2(b) and let the right ISP increase the relative cost of the middle link with the goal of causing the traffic to use the top link. It must decrease the relative cost of the other two links. By doing so, the right ISP may inadvertently cause traffic to use the more expensive bottom link (as the internal costs of the middle ISP in that direction are not directly advertised to the right ISP). The combination of limited gain and uncertainty discourages manipulations.

A second benefit of normalization is that it approximates win-win routing by making path selection sensitive to the concerns of both ISPs. For instance, in a scenario where one ISP has costs in the range [0-10] and the other ISP has costs in the range [0-1000], shortest path routing without normalization will strongly favor the ISP whose costs dominate. Normalization gives the ISPs an equal-footing on which to compare their costs and benefits.

Treating a neighboring ISP as an equal is an approximation to an optimal solution that we have found to work well. This is most likely because of the structure of the ISP marketplace in which peer relationships, at least where there is routing choice, often occur between rough equals. As with peering contracts, however, it would be easy for ISPs to use different weightings if they wanted to account for significant asymmetries, e.g., a higher weight may be assigned to (paying) customers or ISPs that send more traffic. Similarly, instead of computing it based on current advertisements, some ISPs may choose to have a normalization factor based on longer-term averages. Our experiments use an equally weighted normalization based on current advertisements.

4.4 Bounds on the Ratio of Usage Cost

Given that ISPs are encouraged to honestly advertise costs, we would like to also incent them to respect those costs in making their routing choices. In the absence of such an incentive, an upstream ISP might undermine the protocol by selecting locally optimal paths. For instance, the left ISP in Figure 2 might select the bottom interconnection regardless of the advertised costs. It is not straightforward for the downstream ISPs to catch this behavior since their expensive links may be an appropriate choice for some upstream sources.

We use a cost usage ratio to encourage ISPs to honestly select paths. It is inspired by current contractual practices, in which peer ISPs set a traffic exchange ratio that involves no money transfer in the common case [34]. Both upstream and downstream ISPs independently track how the upstream is sending traffic to the downstream.

They keep a running total of usage costs, which we define to be the sum of the advertised cost of a route multiplied by the rate of traffic that is sent along it.

The average usage cost, which is the total usage cost divided by the total rate of traffic, will vary with ISP path selections. With honest path selection, the average usage cost will be low because the upstream ISP tends to avoid paths that are costly for the downstream ISP. In contrast, if the upstream ISP is dishonest, the average usage cost will be higher. Wiser leverages this expected behavior by adding a clause to the contract between ISPs that stipulates a bound on the ratio of the average usage cost to the average advertised cost. Using the same notation as before, we have:

$$\mathsf{avgusage}_{I\rightarrow N} = \frac{\sum_{P,d} \mathsf{advcost}_{N\rightarrow I}(P,d) \times \mathsf{traffic}(P,d)}{\sum_{P,d} \mathsf{traffic}(P,d)}$$

$$\mathsf{avgadv}_{N\rightarrow I} = \frac{\sum_{P,d} \mathsf{advcost}_{N\rightarrow I}}{\sum_{P,d} 1}$$

$$\mathsf{ratio}_{I\rightarrow N} = \frac{\mathsf{avgusage}_{I\rightarrow N}}{\mathsf{avgadv}_{N\rightarrow I}}$$

Above, $\mathsf{avgusage}_{I\rightarrow N}$ is the average usage cost of ISP I sending traffic to ISP N and $\mathsf{avgadv}_{N\rightarrow I}$ is the usage cost that will result if path selections are randomized over the advertised costs of ISP N. The final equation gives the quantity that is checked against the contractual bound which is determined by ISPs based on their situation.

ISPs have the flexibility to make individual decisions within the bound, but are incented to use advertised costs in their overall route selection to stay within the contract. Usage costs also incent an ISP to honestly propagate the received costs in its own advertised costs; if it fails to do so, its upstreams may select paths that are more costly and increase its usage costs. Finally, other contractual clauses are also possible, e.g., a provider might charge a customer based on the measured ratio. We leave their exploration for future work.

5 Implementation

We have implemented Wiser as an extension to BGP on two independent platforms, XORP [54] and SSFNet [46]. We made the following changes to BGP:

• As well as BGP attributes, routing messages carry agnostic costs using a new optional, non-transitive route attribute. (A new community attribute could also be used.) To compute these costs, we use the IGP metric as the internal component. This is similar to the way that ISPs use IGP costs as the basis for MEDs [29, 30]. It would be straightforward to accept costs via a different channel to accommodate ISPs that do not have IGP costs available, e.g., because they use MPLS, or prefer other costs.

• Border routers keep track of the sums of the advertised and received agnostic costs for each neighbor and periodically share them with the other border routers of the same ISP. Information from all the routers is aggregated to compute the normalization factor, which is the ratio of the incoming to outgoing costs summed across all border routers. We leverage the existing iBGP mesh and route reflection mechanisms but new platforms such as RCP [12] can also be easily adapted for this purpose. Intra-ISP partitions do not pose a major problem; routers can either continue using the pre-partition factor or compute it based on the subset of reachable routers.

• When a border router receives routes from a neighbor, it normalizes their advertised costs by multiplying them by the normalization factor for that neighbor. Only normalized costs are propagated within the ISP.

• Each router uses the modified BGP decision process shown in Table 1 to select routes. Compared to BGP, it includes an additional step that selects routes based on the Wiser cost, which is the sum of the normalized received cost and the internal cost. This step comes after considering local preferences that implement commercial policies (e.g., prefer customers over providers). It comes before AS-path, which it effectively replaces, and before internal cost, so that decisions factor in non-local costs.

• When the normalization factor for a neighbor changes significantly (in response to major routing changes), the border routers re-evaluate routes received from that ISP. This is similar to what happens today when IGP costs change. However, unlike IGP cost changes, this re-evaluation can be done in the background, while other tasks are processed with a higher priority. This is because exactly one router, which received the route directly from the neighboring ISP, is responsible for normalizing the costs of any given route. Until that router applies the change, the other routers do not realize that the cost of the route has changed and have a consistent view of it.

• Finally, to verify a neighbor's path selection behavior, border routers log information required to compute the incoming usage costs. This includes the amount of incoming traffic and the announced cost for each destination prefix. A sampling mechanism similar to NetFlow is used for this purpose. Periodically, the estimates are logged to disk and reset. ISPs collect this information from all of their border routers and check if the usage ratio is below the contractual threshold. Similar logging

1. Highest local preference
2. Lowest Wiser cost
3. Shortest AS-path length
4. Lowest origin type
5. eBGP-learned routes over iBGP-learned
6. Lowest IGP cost to egress router
7. Lowest router ID of the BGP speaker

Table 1: *The routing decision process with* Wiser *is similar to that with BGP except for an additional filter (Step 2) that is based on* Wiser *cost.*

is implemented to compute outgoing usage costs, which helps to cross-check if a neighbor claims that this ISP has exceeded the threshold. We have not yet implemented usage cost logging in XORP.

The modifications to BGP above can be deployed incrementally by pairs of ISPs to improve routing between them. Moreover, all routers within an ISP need not be upgraded simultaneously. In a partially upgraded state, the normalization factor can be statically programmed at the upgraded routers, before transitioning to a dynamic computation. (Done this way, some care needs to be taken regarding the consistency of forwarding paths.)

6 Evaluation

Our evaluation of Wiser considers the following questions:

1. How efficient is Wiser *and is it win-win?* For the topologies and cost models that we study, we show that Wiser is win-win and its efficiency comes close to ideal routing that globally optimizes the internetwork based on complete information.

2. What is the overhead of running Wiser*?* We show that the overhead of Wiser is similar to BGP in terms of implementation complexity, routing message and computational requirements. Its convergence time is acceptable even in response to major failures.

3. How robust is Wiser *to cheating?* For the strategies that we study, we show that Wiser is robust in that it limits the gain for dishonest ISPs and the loss for honest ISPs.

4. What factors facilitate efficient routing with Wiser*?* While previous evaluations use inputs based on the current Internet, this part uses synthetic cost and topology models to understand situations under which Wiser will be effective. We show that it produces efficient end-to-end paths as long as ISP objectives include relevant factors and that its efficiency is higher when the costs of ISPs that interconnect in multiple places are similar.

The answer to the questions above depend on many aspects of ISP networks, some of which are hard to model. To focus on realistic rather than theoretical best- or worst-case bounds, we combine measured data with models based on known properties of the Internet. As measured input, we use an internetwork topology of 65 ISPs and their interconnections [44]. These ISPs have diverse sizes and geographical presence. A node in an ISP topology corresponds to a city where the ISP has a point of presence (PoP). We compute paths over these topologies (rather than use separately measured paths) so that our results are less influenced by measurement errors. The models that we use depend on the experiment but a common one is approximating propagation delay of a link by the geographic distance between the two end-point cities [35]. Our results are of course limited to the topologies, models, and ISPs behaviors that we study.

Efficiency (§6.1)	Similar ISP objectives	Path length
		Bandwidth provisioning
	Diverse ISP Objectives	Path length and bandwidth
		Inferred weights
Overhead (§6.2)	Implementation complexity	
	Routing msgs. and convergence	Load independent costs
		Load sensitive costs
	Computational requirements	Normal workloads
		Highly dynamic workloads
Robustness to cheating (§6.3)	Dishonest cost disclosure	
	Dishonest path selection	
	Dishonest cost propagation	
Understanding efficiency (§6.4)	Impact of topology	
	Impact of ISPs' objectives	
	Comparison with Nexit	
	ISP costs for individual flows	

Table 2: *Experiments with* Wiser*. The shaded cells correspond to experiments not presented in this paper.*

To compute the routing produced by Wiser, we use our XORP and SSFNet prototypes as well as a custom, high-level simulator that does not model message passing. These three engines have different scaling properties because they model different levels of detail. We use the custom simulator to study efficiency and robustness to cheating, and the prototypes to study overhead.

Table 2 provides an overview of the experiments we have conducted. We present only a subset of results in the following sections due to space limitations.

6.1 Efficiency

The efficiency of global routing can be measured in several ways. We study scenarios with both similar and diverse ISP objectives. For the former, we first consider end-to-end path length, since long paths degrade application performance. This metric assumes that the network capacity is well-matched to traffic and ISPs are primarily interested in minimizing the distance a packet travels inside their networks. We then consider bandwidth provisioning required by ISPs to avoid overload when the traffic and capacity are no longer well-matched, e.g., due to a failure. We use inferred link weights for scenarios with diverse ISP objectives, since they capture the choices of different ISPs.

6.1.1 Path length

We compare the path lengths produced by Wiser to two other routing methods: *global* and *unilateral*. The former minimizes path length using global information on the lengths of network links. It is not feasible in practice due to ISP autonomy issues, and we study it to understand the cost of this autonomy. *Unilateral* mimics the

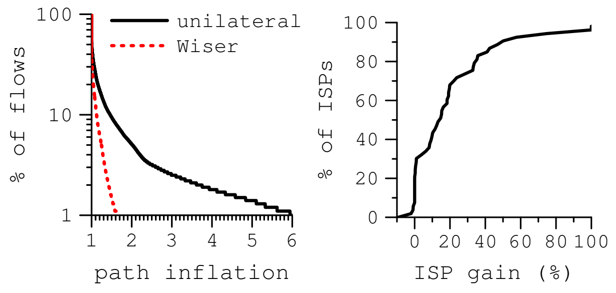


Figure 3: Wiser *produces efficient routing paths and is win-win.* Left: *The CCDF (in log scale) of path inflation.* Right: *The CDF of gain for individual ISPs with* Wiser.

common BGP policies of shortest AS-path and early-exit. With Wiser, ISPs use internal distance as the basis for assigning costs to internal paths. This is a rough measure of the resources consumed inside the network; minimizing it is the motivation for early-exit routing. All three routing methods follow common commercial policies [34, 48], e.g., ISPs do not provide transit to peers and providers. Traffic in this experiment consists of a flow between each pair of PoPs.

We find that, unlike *unilateral*, end-to-end path lengths with Wiser come close to that of *global*. The average path length with Wiser is only 4% higher than *global*, while it is 13% higher with *unilateral*. This improvement is useful, though it suggests that the common case path length in the Internet today is acceptable.

We find the more significant difference between *unilateral* and Wiser to be the distribution of path lengths of individual flows. The left graph in Figure 3 shows the path length inflation with Wiser and *unilateral* compared to *global* as a complimentary cumulative distribution function (CCDF): the y-value is the percentage of flows inflated by at least the corresponding x-value. An inflation of two implies that the path length was doubled.

We see that, while half of the paths are not inflated at all, some are highly inflated with *unilateral*: 5% are inflated by more than a factor of 2 and 1% by more than a factor of 6. In terms of absolute inflation, 5% of the paths are inflated by over 40 ms and 1% by over 70 ms. High inflation is not limited to short paths or intercontinental paths: even when we consider only paths that are longer than 20 ms with unilateral routing or only paths within the USA, the worst 1% are still inflated by a similar factor. Applications using overly long paths will experience high latencies unless the paths are (manually) fixed.

To better understand how overly long paths arise, consider two examples. The first example involves two large ISPs in the USA that have national presence but interconnect largely along the two coasts. For traffic going from the middle of the country in one ISP to the east coast inside another, with *unilateral*, the source ISP picks an interconnection that is closest to the source. It happens to be an interconnection on the west coast. Hence, the source ISP takes the traffic to the west coast and the destination ISP brings it to the east coast. With *global*, an east coast interconnection is employed, leading to a path that is roughly three times shorter.

The second example involves traffic going from the southeast to the east coast of the USA between two ISPs that do not directly interconnect. With *unilateral*, the source ISP transfers the packets to an intermediate ISP that does not interconnect with the destination ISP on the east coast, which makes the traffic traverse an interconnection on the west coast before returning to the east coast. With *global*, the chosen intermediate ISP interconnects with the destination ISP at the east coast itself. The resulting path is roughly five times shorter.

The graph shows that Wiser can automatically fix such overly long paths: 5% of the paths are inflated by a factor of 1.2 and the worst 1% by only a factor of 1.5. This gain in efficiency stems directly from joint control over routing. Unlike *unilateral*, by combining inputs from all ISPs, Wiser can avoid long paths that, while slightly favorable for the source ISP, are very long end-to-end.

The right graph in Figure 3 shows that improvement in end-to-end paths with Wiser does not require individual ISPs to suffer for the global good, i.e., it is close to win-win as we desire. It plots the CDF of gain for individual ISPs with Wiser, measured as the average reduction in distance, relative to *unilateral*, that a packet travels inside the ISP's network. In terms of adoption incentives, this measure ignores the improvement in performance for customers and the reduction in operational cost from not having to manually fix poor routes. Almost no ISP loses by running Wiser and many ISPs gain, and thus ISPs have an adoption incentive. The graph shows that a handful of ISPs do lose a little. These are small, edge ISPs for whom the changed routing pattern represents a minor loss according to our measure. We find that the overall quality of routing is not impacted even if such ISPs chose to not run Wiser. Alternatively, they can also negotiate a different normalization factor with their neighbors.

6.1.2 Bandwidth provisioning

To be free of congestion under dynamic conditions, ISPs can either highly provision their networks or dynamically alleviate overload. The former approach is common today because ISPs do not have proper control over routing. But Wiser can help ISPs with the latter, and thus reduce provisioning, by signaling to their neighbors to send less traffic along certain paths.

To assess this benefit, we must use models based on known properties of Internet routing. Provisioning is hard to evaluate because it is affected by factors such as link capacities and workloads for which there is almost no public information. We use models that are similar to previous work [21, 28]. We use a gravity approach to model

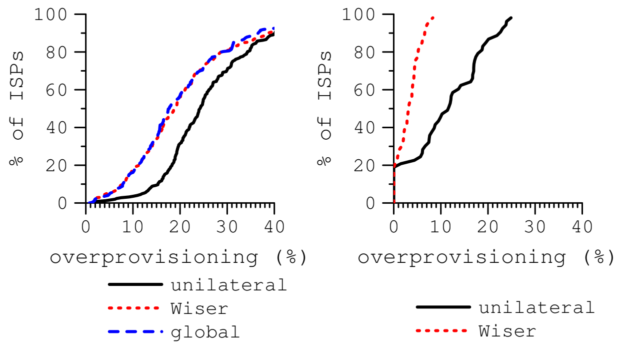


Figure 4: Wiser *reduces overprovisioning level.* Left: *Pairs of ISPs.* Right: *The entire internetwork topology.*

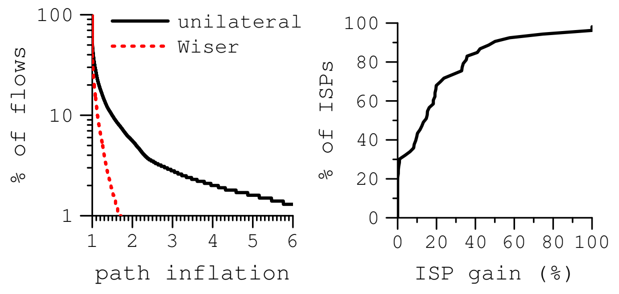


Figure 5: Wiser *produces efficient routing paths with inferred link weights.* Left: *The CCDF of path inflation. Right: The CDF of gain for individual ISPs with* Wiser.

traffic between two PoPs as proportional to the product of the population of their host cities [49, 55]. We model link capacities as proportional to the stable load on the link, since in steady-state a well-designed network tends to be roughly matched to its traffic [55]. We simulate dynamic conditions by failing interconnections between ISPs, as these failures can cause congestion today [23, 24]; we leave the study of other perturbations such as internal failures for future work.

We measure efficiency using the *overprovisioning level*. For a link, this is the maximum additional load, compared to its stable load, that it carries across all simulated failures. The overprovisioning level for an ISP is the weighted average of the overprovisioning levels of its links. The weight of a link is its stable load, to reflect that doubling the capacity is costlier for higher-capacity links.

We compare the same three routing methods. *Global* minimizes the overprovisioning level across the entire internetwork and is computed by solving a linear programming problem. For computational tractability, we allow fractional routing which provides a lower-bound on any protocol with non-fractional routing. To illustrate the benefit of Wiser, we set costs to be the product of static and dynamic factors. The static factor is its length. The dynamic factor varies linearly with load but is updated only when the load changes by more than 10%. *Unilateral* is computed as before and is load-insensitive; it is uncommon for ISPs to automatically respond to overload because that may hurt neighbors. Techniques that let ISPs respond without hurting others have limited efficacy [11].

Due to computational limits, we could not compute *global* for the entire topology. We divide our results into two parts. First, we compare all three routing methods over subsets of the overall topology. This illustrates how close to *global* Wiser can get. Then, we compare *unilateral* and Wiser over the entire topology.

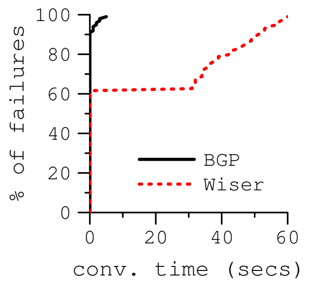
The left graph in Figure 4 shows the results for subset topologies which are pairs of adjacent ISPs with traffic flowing between all pairs of PoPs in the two ISPs. There are two points for each ISP pair. We find that, unlike *unilateral*, Wiser closely approximates *global* to the extent that their lines are visually indistinguishable. Relative to the *global*, the average overprovisioning level is 0% with Wiser and 7% with *unilateral*. The right graph shows the results for the entire topology. Traffic consists of flows between a randomly selected 10% of all possible PoP pairs. We simulate the failure of each interconnection between tier-1 ISPs. There are over 400 such interconnections in the dataset. As is the case for the smaller topologies, the overprovisioning with Wiser is much less than that with *unilateral*. The average difference is 8%. While ISPs usually upgrade capacity by bigger factors, this difference may translate into significant monetary savings if they need to upgrade less often.

Though not shown in the graphs, we find that Wiser is win-win for this measure as well, i.e., the overprovisioning level of individual ISPs does not increase compared to *unilateral*.

6.1.3 Inferred link weights

Finally, we study the efficiency of Wiser when ISPs have diverse objectives. We model this using link weights inferred for each ISP [44]. The shortest path routing produced by these weights is consistent with the routing observed inside ISPs, though these weights are not necessarily identical to what the ISP uses [44]. We assume that these weights capture the existing internal objectives of ISPs at the time the topologies and weights were inferred.

The left graph in Figure 5 shows the efficiency of Wiser and *unilateral* with inferred link weights. It plots the multiplicative inflation in path length relative to the length of *global* (as computed in §6.1.1). Unlike *unilateral*, Wiser comes close to *global*. The worst 1% of the paths are inflated by a factor of 7 with *unilateral* but by only 1.7 with Wiser. The right graph shows that Wiser is win-win for this cost model as well. It plots the gain for individual ISPs, measured as the average reduction in path *weight* relative to *unilateral*.



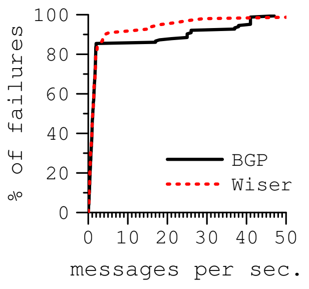


Figure 6: *The convergence time and routing message overhead of* Wiser. Left: *The CDF of convergence time.* Right: *The CDF of maximum rate of routing messages.*

6.2 Overhead

We now study the overhead of Wiser relative to BGP using our XORP and SSFNet prototypes.

Implementation complexity Using lines of code as a rough measure of implementation complexity, we find that our XORP and SSFNet prototypes add only 3% and 6% lines to their respective BGP implementations.

Convergence time and routing messages We evaluate the convergence time and routing message overhead of Wiser in response to routing perturbations. We use SSFNet for these experiments. Due to memory limitations, we consider only the "core" of the internetwork topology, which includes roughly 300 nodes that belong to tier-1 ISPs and have multiple neighbors. We believe that our results are reflective of the overall topology because our measures depend heavily on the core [25].

We perturb routing with failures of the interconnections between tier-1 ISPs as in §6.1.2. These are significant events, and most changes will have a smaller impact on routing. One interconnection fails per trial. All costs are static, which corresponds to overprovisioned networks.

The left graph in Figure 6 plots the CDF of the time it takes for the routing to converge after the link fails. There is one point for each simulated failure. Connectivity is restored in both Wiser and BGP as soon as the failure is discovered but the convergence time can differ. For 60% of the failures, the convergence time of Wiser is similar to that of BGP. It is higher for 40% of them.

BGP routers advertise only reachability, and so the routing converges soon after the routers attached to the failed link withdraw routes that use that link and announce new routes. With Wiser, if the normalization factor changes significantly, as it may for a major failure, more routing messages can follow the initial announcements. The delay in this case is dominated by the MRAI (minimum route advertisement interval with a default of 30 seconds) timer of BGP which determines the minimum gap between routing messages sent to neighbor.

These convergence times seem acceptable to us for major changes, especially since connectivity is restored quickly. Even faster convergence could be obtained with lower values of MRAI, as advocated by some [15], or by ignoring MRAI for normalization factor changes.

The right graph in Figure 6 plots the CDF of the maximum message rate experienced by any router in the topology. We count messages from when the link fails to when the routing converges. We see that the routing message overhead of Wiser is comparable to that of BGP. It is slightly lower probably due to its longer convergence time for a subset of the cases.

Computational requirements We use XORP to study the computational requirements of Wiser for typical workloads. Routers that run Wiser need to track normalization factors and usage costs, as well as BGP-related responsibilities. To measure this added burden, we feed in a log of routing messages collected by a RouteViews server from forty-one diverse routers in the Internet to a machine (2.2 GHz, 3.8GB RAM) that runs Wiser. Because the logs do not contain Wiser costs, we attach a randomly generated cost in the integral range [1..10] to each message. So that the routing tables fit in memory, we randomly select ten out of the forty-one message sources in a trial. We conduct five trials each for logs from two different days and find that the computational load of Wiser is only 15-25% higher than BGP.

6.3 Robustness to Cheating

To study potential for abuse, we consider a form of cheating that we consider plausible: a dishonest ISP manipulates its advertised costs and favors its own costs in path selection to try to reduce the average internal cost of the traffic it carries. There are bound to be other motivations to cheat and forms of misbehavior of which we are not aware, but we must leave these for future study.

In our experiment, we consider pairs of ISPs that interconnect in multiple places. This allows us to study the average cost of carrying traffic in isolation because the overall traffic in each ISP network stays constant. Traffic is composed of a unit flow between each pair of PoPs. ISPs aim to minimize internal distance. One ISP in the pair is honest and the other is dishonest. We assume that the dishonest ISP has complete information about traffic and the other ISP's internal costs of sending traffic (which are never directly disclosed). This overestimates the abilities of the cheater because there will be uncertainty in this information in practice.

The dishonest ISP changes the relative values of advertised costs and uses a reduced normalization factor to favor its internal costs when it selects paths. Computing advertised costs that give the dishonest ISP the most gain within the normalization constraint is NP-hard (because it similar to bin packing). We use hill climbing to approximate these costs. Similarly, we use binary search to find the lowest normalization factor that still satisfies the

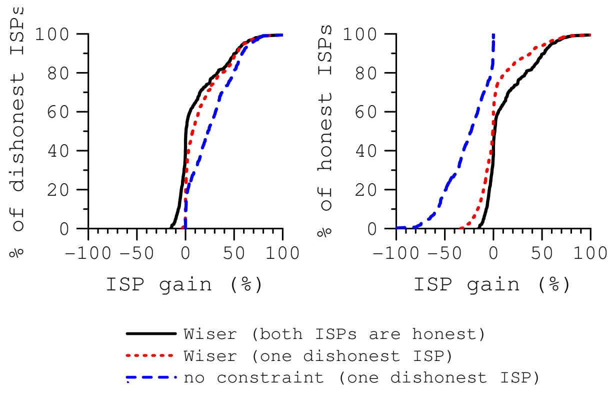


Figure 7: Wiser *limits the gain for dishonest ISPs and the loss for honest ISPs.* Left: *The CDF of gain for dishonest ISPs.* Right: *The CDF of gain for honest ISPs.*

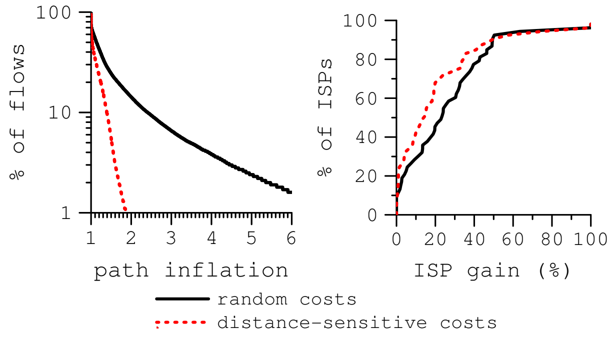


Figure 8: *Efficiency of* Wiser *and gain for individual ISPs with two synthetic cost models.* Left: *The CCDF of path inflation. Right: The CDF of gain for individual ISPs.*

bound on the usage cost ratio. We choose a value of 0.8 for this bound for illustrative purposes.

Figure 7 shows the impact of cheating with this strategy. The graphs plot the CDF of gain for the dishonest and honest ISPs, where the gain is measured as the reduction in average distance relative to *unilateral*. For comparison, we also plot the gains for a scenario where both ISPs honestly implement Wiser and for a scenario where the dishonest ISP can cheat arbitrarily because the normalization and usage cost ratio constraints do not exist. We see that with Wiser the curves for one dishonest ISP are close to the case where both ISPs are honest. This implies that Wiser limits the gain for the dishonest ISP and the loss for the honest ISP. We also see higher gain and loss when the constraints imposed by Wiser are removed, which further indicates their effectiveness.

6.4 Understanding the Efficiency of Wiser

We now switch from evaluating Wiser over realistic inputs to explore in more general terms the ISP cost models and topologies for which it can provide efficient routing.

6.4.1 Impact of ISP cost models

We experiment with synthetic cost models to understand under what circumstances efficient end-to-end paths are produced. We know that Wiser produces efficient routes with inferred link weights that model ISPs' costs today. But it will not necessarily do so for arbitrary models of internal ISP costs. To probe this issue, we first consider a scenario where we assume that each ISP has an unknown (to us) objective, and randomly assign costs to each link from a finite range. The solid curves in Figure 8 show the results. The left graph plots the CCDF of path inflation relative to *global*, as computed in § 6.1.1. The right graph plots the individual gain for ISPs, measured as the average reduction in cost of carrying traffic. The graphs show that the quality of end-to-end paths with random costs assignment is poor even though Wiser individually benefits all ISPs.

However, ISPs objectives are not arbitrary but influenced by measures of interest to them and their users, e.g., all reasonable objectives are likely to reflect path length to some extent. To evaluate Wiser in this more realistic scenario, we assume that the cost of a link inside an ISP is $\frac{1}{2}L + rL$, where L is the length of the link and r, which is a random number in the range $[0..1]$, captures the unknown components of the ISP objective, scaled to match the length component. Figure 8 shows that the end-to-end paths with these "distance-sensitive costs" are efficient and ISPs individually benefit as well.

6.4.2 Impact of topology

We now use an analytic model to understand the topological characteristics that make Wiser effective. To make this tractable, we restrict ourselves to the two-ISP base case. Generalizing the arguments presented below lead to similar inferences for the multi-ISP case [26].

Consider two ISPs, ISP-1 and ISP-W, that interconnect in N places. We model the internal ISP topology as a fully-connected mesh in which the cost of transporting a packet between two nodes is drawn from a uniform random distribution in the range [0..1] for ISP-1 and $[0..W]$ for ISP-W ($W \geq 1$). W captures ISP heterogeneity: a higher W stems from a higher average cost of carrying packets inside ISP-W. The cost of transporting a packet across the two ISPs is the sum of the costs incurred inside each. This assumes that the ISPs' costs have been mapped to comparable units. The expected costs in this model with different routing methods is shown in Table 3. Their derivation is outlined in [26]. Our model is simplistic, e.g., paths between pairs of nodes are not truly independent of other paths, but we find its results to be consistent with our other experiments. It is also arguably more realistic than the only other analytic model of two-ISP routing of which we are aware [19] because it captures factors such as N and W that we show to be important.

We study the efficiency of the different routing methods as a function of W. We use cost inflation relative

Uni-lateral	$C_{unilateral}(N,W) = \frac{(W+1)(3+N)}{4(N+1)}$ $C^1_{unilateral}(N,W) = \frac{N+3}{4(N+1)}$
Global	$C_{global}(N,W) = \frac{N}{3W}\ {}_2F1(\frac{3}{2}, 1-N, \frac{5}{2}, \frac{1}{2W}) + \frac{N}{2(2W)^N}\left(\frac{(2W+1)((2W-1)^N-1)}{N} - \frac{(2W-1)^{N+1}-1}{N+1}\right) + \frac{2WN+W+1}{(2N+1)(2W)^N}$ $C^1_{global}(N,W) = \frac{N}{6W}\ {}_2F1\left(\frac{3}{2}, 1-N, \frac{5}{2}, \frac{1}{2W}\right) + \frac{(2W-1)^N-1}{2(2W)^N} + \frac{N+1}{(2N+1)(2W)^N}$ where ${}_2F1(a,b,c,z) = \sum_{k=0}^{\infty} \frac{\binom{a}{k}\binom{b}{k}z^k}{\binom{c}{k}k!}$.
Wiser	$C_{Wiser}(N,W) = \frac{(W+1)C_{global}(1)}{2}$ $C^1_{Wiser}(N,W) = C_{global}(N,1)/2$

Table 3: *Expected routing costs with various routing methods.* $C_m(N,W)$ *is the total cost of routing using method* m*, and* $C^1_m(N,W)$ *is the cost incurred by ISP-*1*. The cost incurred by ISP-*W *is the difference of the two.*

to *global* as the measure of efficiency. This captures the average inflation, not the worst-case inflation. As such, it underestimates the benefit of Wiser by discounting the impact of egregiously bad cases.

The left graph in Figure 9 plots cost inflation as a function of W, where we have selected $N = 6$ to provide an example. Wiser is always more efficient than *unilateral*. It comes close to *global* for low values of W but is less efficient as W increases. To investigate this effect, the right graph plots the gain of individual ISPs with Wiser and *global*. Gain is computed as the reduction in cost relative to *unilateral*. Both ISPs gain equally with Wiser, but ISP-W gains at the expense of ISP-1 with *global*. Because ISP-W's costs are higher, globally optimal will sacrifice ISP-1's interests to the greater good. Thus, Wiser enables ISPs to cooperate without losing but the overall efficiency is less when ISPs' costs are very diverse. That the efficiency of Wiser comes close to that of *global* for realistic topologies (§6.1) suggests that the costs of ISPs that interconnect in multiple places are roughly similar for the metrics that we study.

6.5 Summary of Results

We end this section with a summary of results. For the topologies and metrics we studied, we showed that joint control of routing in Wiser comes close to ideal routing that globally optimizes end-to-end paths based on complete information. For the path length metric, while the worst 1% of that paths are inflated by a factor of 6 with today's routing practices, they are inflated only by a factor of 1.5 with Wiser. Wiser also reduces the bandwidth provisioning required by ISPs to handle the dynamic conditions that we simulated by 8% on average. We explored other ISP objectives and found that Wiser continues to

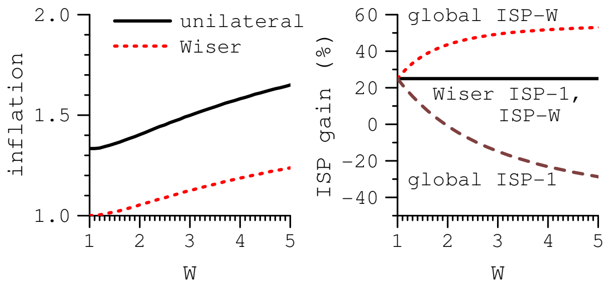


Figure 9: *Cost inflation (left) and gain for individual ISPs (right) as a function of* W *with* N*=6.*

produce efficient paths as long as ISPs' objectives include factors that influence end-to-end paths, as is typically the case today. We found the overhead of Wiser to be similar to BGP in terms of implementation complexity, routing message and computational requirements. For the strategies that we studied, cost normalization and usage cost ratio constraints limit the gain for dishonest ISPs and the loss for honest ISPs. Finally, our analysis showed that the efficiency of Wiser depends on the similarity in ISPs' costs, but it is always higher than unilateral routing and, unlike optimal routing, stays win-win.

7 Discussion

The most surprising aspect of our work is perhaps that the simple mechanisms of Wiser are so effective in our experiments. Wiser obtains high levels of efficiency by combining costs over neighboring pairs of ISPs. This suggests that it is neither necessary nor particularly advantageous to construct more complex costs that are meaningful across larger groups of ISPs, e.g., global currency. This is somewhat surprising because currencies with a larger scope could allow better multi-way trades, in the same way that bigger markets tend to be more efficient. But it confers a significant practical advantage: It is much simpler to implement costs between pairs of ISPs because this mirrors the contractual structure of the Internet.

To see whether even simpler approaches would be equally efficient, we studied two alternate approaches. First, we ran Wiser with ordinal (rather than cardinal) costs to disclose less information. This is an interesting point in the design space because MEDs have ordinal semantics. We found that efficiency with Wiser using ordinal costs was little better than unilateral routing. Cardinal costs can lead to greater efficiency by using relative magnitudes to identify path changes that lead to a small loss for one ISP but a bigger gain for another. Second, we ran a variant of Wiser for pairs of flows. Wiser takes a holistic view of the traffic exchanged between two ISPs and is efficient; at the other extreme, unilateral routing considers each flow in isolation and can be inefficient. Pairs of flows that go in opposite directions are a natural intermediate point. However, we found the efficiency of this approach to be little better than unilateral routing.

Taken together, the observations above suggest that Wiser occupies an attractive point in the design space: more complex approaches gain little efficiency; and simpler approaches lose efficiency.

Finally, one aspect of our design that we have mostly deferred to future work is stability with different cost functions. Provably stable dynamic routing in large networks is an open research question even under non-strategic behavior [6, 22, 41, 52]. Existing research provides guidelines for assigning costs in a way that enhances routing stability [3, 6, 41, 53] and which apply to our setting. We also note that information sharing and non-greedy decision making may enhance stability because it discourages ISPs from making changes that adversely affect each other.

8 Related Work

Much recent work has highlighted that BGP provides poor control over routing and computes inefficient paths [4, 11, 19, 36, 40, 44, 50]. Our work shows how these problems can be addressed while being consistent with ISP interests. Stability of BGP under different commercial policies has also been scrutinized [16]. Wiser finds paths within commercial preferences and thus neither helps nor hurts on that account except for removing MED-induced oscillations [29].

Wiser builds on our earlier work on Nexit, a framework by which two ISPs can negotiate routes [28]. Unlike Nexit, Wiser handles the general case of multiple ISPs and has a much lower overhead. It accomplishes this while preserving ISP interests in the same manner and disclosing similar amount of information.

Other work has explored optimizing interdomain routing using existing BGP knobs [11, 17, 37, 51]. While this helps, it has limited effectiveness because ISPs lack visibility outside their own network and have little incentive to suffer for the greater good. Wiser tackles these root problems directly. At the other extreme, work on interdomain quality-of-service (QoS) requires full cooperation between ISPs, including the disclosure of sensitive information and agreement on the optimization metric [10]. Wiser eschews this high degree of cooperation to preserve ISP interests and facilitate deployment.

Researchers have also explored monetary payments for carrying traffic along specified routes [2, 32]. This requires ISPs to disclose monetary costs at a fine granularity; approaches based on mechanism design [13] can encourage the disclosure of true costs via strategy-proof (but not budget-balanced) mechanisms. Regardless, monetary costs are difficult to compute [42], can leak sensitive information that can be used to undercut the market outside of the mechanism, and are not compatible with the current "customer pays" charging model that is independent of the direction of traffic. Wiser retains the current charging model to be practical, and our results also suggest that monetary costs are not necessary for better efficiency.

Finally, Wiser is similar in spirit to other work on competitive interests. Like BitTorrent [9], Wiser uses bilateral coordination and favors practicality over the prevention of cheating [43]. Like work on load management in federated systems [5], Wiser leverages offline and bilateral contracts to simplify online operation.

9 Conclusions

Our work shows that, at least for Internet routing, competing interests can be harnessed using practical protocols and without significant loss in efficiency. Wiser enables ISPs to jointly control routing and find good end-to-end, policy-compliant paths while allowing them to improve their own routing by their own reckoning. It builds on the existing contractual framework, does not require new monetary exchange, and is incrementally deployable. We evaluated Wiser via simulation over measured topologies and with XORP and SSFNet prototypes. In our experiments, Wiser was win-win and its efficiency came close to that of routes that were globally optimized with complete information. It was especially useful in automatically improving the tail of the paths which can be overly long with current routing methods. The overhead of Wiser was similar to BGP, and the built-in checks and balances encouraged ISPs to use it honestly.

Our evaluation suggests that Wiser is a promising way to provide more control to ISPs while increasing the efficiency of routing at the same time. To better understand its benefit in practice, in the future, we will conduct a more detailed investigation into models of ISPs' behaviors and the extent and frequency of routing inefficiencies that can be fixed with Wiser.

We hope that lessons from our work will prove useful in other contexts. Normalization may be broadly useful because it enables trading where there is no common basis for assigning values to commodities. Similarly, mechanisms such as usage cost ratios that reduce the degrees of freedom of individual parties may help to address other conflicts between efficiency and incentive compatibility. And the use of offline contractual clauses, which has received little attention in research, appears to be a powerful method to simplify online operation.

Acknowledgments

We thank Benson Schliesser (Savvis Inc.), the anonymous referees, and our shepherd, Albert Greenberg, for their feedback. This work was supported in part by NSF Grant CNS-0435065.

References

[1] M. Afergan. Using repeated games to design incentive-based routing systems. In *INFOCOM*, 2006.

[2] M. Afergan and J. Wroclawski. On the benefits and feasibility of incentive based routing infrastructure. In *PINS Workshop*, 2004.

[3] G. Apostolopoulos, *et al.* Quality of service based routing: A performance perspective. In *SIGCOMM*, 1998.

[4] D. O. Awduche, *et al.* Overview and principles of Internet traffic engineering. RFC-3272, 2002.

[5] M. Balazinska, H. Balakrishnan, and M. Stonebraker. Contract-based load management in federated distributed systems. In *NSDI*, 2004.

[6] A. Basu, A. Liu, and S. Ramanathan. Routing using potentials: A dynamic traffic aware routing algorithm. In *SIGCOMM*, 2003.

[7] S. J. Brams. *Negotiation Games: Applying game theory to bargaining and arbitration*. Routeledge, 1990.

[8] D. Clark. The design philosophy of the DARPA Internet protocols. In *SIGCOMM*, 1988.

[9] B. Cohen. Incentives build robustness in BitTorrent. In *1st Workshop on Economics of Peer-to-Peer Systems*, 2003.

[10] E. S. Crawley, *et al.* A framework for QoS-based routing in the Internet. RFC-2386, 1998.

[11] N. Feamster, J. Borkenhagen, and J. Rexford. Guidelines for interdomain traffic engineering. *CCR*, 33(5), 2003.

[12] N. Feamster, *et al.* The case for separating routing from routers. In *FDNA Workshop*, 2004.

[13] J. Feigenbaum, *et al.* A BGP-based mechanism for lowest-cost routing. In *PODC*, 2002.

[14] B. Fortz and M. Thorup. Internet traffic engineering by optimizing OSPF weights. In *INFOCOM*, 2000.

[15] T. Griffin and B. Premore. An experimental analysis of BGP convergence time. In *ICNP*, 2001.

[16] T. Griffin and G. T. Wilfong. An analysis of BGP convergence properties. In *SIGCOMM*, 1999.

[17] Internap Flow Control Xcelerator. http://www.internap.com/product/technology/fcx/.

[18] P. Jacob and B. Davie. Technical challenges in the delivery of interprovider QoS. *IEEE Communications Magazine*, 43(6), 2005.

[19] R. Johari and J. N. Tsitsiklis. Routing and peering in a competitive Internet. T.R. P-2570, MIT LIDS, 2003.

[20] K. Johnson, *et al.* The measured performance of content distribution networks. In *Int'l Web Caching and Content Delivery Workshop*, 2000.

[21] S. Kandula, *et al.* Walking the tightrope: Responsive yet stable traffic engineering. In *SIGCOMM*, 2005.

[22] A. Khanna and J. Zinky. The revised ARPANET routing metric. In *SIGCOMM*, 1989.

[23] B. Kruglov. Re: Cogent and level3 peering issues. NANOG mailing list archives: http://www.merit.edu/mail.archives/nanog/2002-12/msg00379.html, 2002.

[24] S. M. Kusiak. Re: Congestion peering C&W <-> @home. NANOG mailing list archives: http://www.merit.edu/mail.archives/nanog/2001-11/msg00282.html, 2001.

[25] C. Labovitz, *et al.* An experimental study of delayed Internet routing convergence. In *SIGCOMM*, 2000.

[26] R. Mahajan. *Practical and Efficient Internet Routing with Competing Interests*. Ph.D. thesis, University of Washington, 2005.

[27] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, 2002.

[28] R. Mahajan, D. Wetherall, and T. Anderson. Negotiation-based routing between neighboring domains. In *NSDI*, 2005.

[29] D. McPherson and V. Gill. BGP MED considerations. Internet draft, 2005.

[30] CA*net routing policy. http://www.canarie.ca/canet4/services/c4_routing_policy.pdf, 2003.

[31] R. Miller. Legal battle ended for AT&T, MCI. InternetNews.com, 2004. http://www.internetnews.com/xSP/article.php/3316751.

[32] R. Mortier and I. Pratt. Incentive based inter-domain routeing. In *ICQT Workshop*, 2003.

[33] R. B. Myerson and M. A. Satterthwaite. Efficient mechanisms for bilateral trading. *Journal of Economic Theory*, 29(2), 1983. Cited in Brams [7].

[34] W. B. Norton. Internet service providers and peering. Equinix whitepaper, version 2.5, 2001. http://www.equinix.com/pdf/whitepapers/PeeringWP.2.pdf.

[35] V. N. Padmanabhan and L. Subramanian. An investigation of geographic mapping techniques for Internet hosts. In *SIGCOMM*, 2001.

[36] B. Quoitin, *et al.* Interdomain traffic engineering with bgp. *IEEE Communications Magazine*, 41(5), 2003.

[37] B. Quoitin, *et al.* Interdomain traffic engineering with redistribution communities. *Computer Communications Journal*, 27(4), 2004.

[38] H. Raiffa. *The art and science of negotiation*. Harvard University Press, 1982.

[39] T. Roughgarden and E. Tardos. How bad is selfish routing? *Journal of the ACM*, 49(2), 2002.

[40] S. Savage, *et al.* The end-to-end effects of Internet path selection. In *SIGCOMM*, 1999.

[41] A. Shaikh, J. Rexford, and K. G. Shin. Load-sensitive routing of long-lived IP flows. In *SIGCOMM*, 1999.

[42] S. Shenker, *et al.* Pricing in computer networks: Reshaping the research agenda. *CCR*, 26(2), 1996.

[43] J. Shneidman, D. C. Parkes, and L. Massoulie. Faithfulness in Internet algorithms. In *PINS Workshop*, 2004.

[44] N. Spring, R. Mahajan, and T. Anderson. Quantifying the causes of path inflation. In *SIGCOMM*, 2003.

[45] N. Spring, *et al.* Measuring ISP topologies with Rocketfuel. *IEEE/ACM ToN*, 12(1), 2004.

[46] Scalable simulation framework. http://www.ssfnet.org/.

[47] S. Stickland. Utilising upstream MED values. NANOG mailing list archives: http://www.merit.edu/mail.archives/nanog/2005-03/msg00400.html, 2005.

[48] L. Subramanian, *et al.* Characterizing the Internet hierarchy from multiple vantage points. In *INFOCOM*, 2002.

[49] N. Taft, *et al.* Understanding traffic dynamics at a backbone PoP. In *SPIE ITCOM*, 2001.

[50] H. Tangmunarunkit, *et al.* The impact of routing policy on Internet paths. In *INFOCOM*, 2001.

[51] S. Uhlig and O. Bonaventure. Designing BGP-based outbound traffic engineering techniques for stub ASes. *CCR*, 34(5), 2004.

[52] Z. Wang and J. Crowcroft. Analysis of shortest-path routing algorithms in a dynamic routing network environment. *CCR*, 22(2), 1992.

[53] Z. Wang and J. Crowcroft. Quality-of-service routing for supporting multimedia applications. *IEEE JSAC*, 14(7), 1996.

[54] XORP: Open source IP router. http://www.xorp.org/.

[55] Y. Zhang, *et al.* Fast accurate computation of large-scale IP traffic matrices from link loads. In *SIGMETRICS*, 2003.

Tesseract: A 4D Network Control Plane

Hong Yan[†], *David A. Maltz*[‡], *T. S. Eugene Ng*[§], *Hemant Gogineni*[†], *Hui Zhang*[†], *Zheng Cai*[§]
[†]*Carnegie Mellon University* [‡]*Microsoft Research* [§]*Rice University*

Abstract

We present Tesseract, an experimental system that enables the *direct control* of a computer network that is under a single administrative domain. Tesseract's design is based on the 4D architecture, which advocates the decomposition of the network control plane into *decision*, *dissemination*, *discovery*, and *data* planes. Tesseract provides two primary abstract services to enable direct control: the *dissemination service* that carries opaque control information from the network decision element to the nodes in the network, and the *node configuration service* which provides the interface for the decision element to command the nodes in the network to carry out the desired control policies.

Tesseract is designed to enable easy innovation. The neighbor discovery, dissemination and node configuration services, which are agnostic to network control policies, are the only distributed functions implemented in the switch nodes. A variety of network control policies can be implemented outside of switch nodes *without* the need for introducing new distributed protocols. Tesseract also minimizes the need for manual node configurations to reduce human errors. We evaluate Tesseract's responsiveness and robustness when applied to backbone and enterprise network topologies in the Emulab environment. We find that Tesseract is resilient to component failures. Its responsiveness for intra-domain routing control is sufficiently scalable to handle a thousand nodes. Moreover, we demonstrate Tesseract's flexibility by showing its application in joint packet forwarding and policy based filtering for IP networks, and in link-cost driven Ethernet packet forwarding.

1 Introduction

We present Tesseract, an experimental system that enables the *direct control* of a computer network that is under a single administrative domain. The term direct control refers to a network control paradigm in which a *decision element* directly and explicitly creates the forwarding state at the network nodes, rather than indirectly configuring other processes that then compute the forwarding state. This paradigm can significantly simplify network control.

In a typical IP network today, the desired control policy of an administrative domain is implemented via the synthesis of several indirect control mechanisms. For example, load balanced best-effort forwarding may be implemented by carefully tuning OSPF link weights to indirectly control the paths used for forwarding. Inter-domain routing policy may be indirectly implemented by setting OSPF link weights to change the local cost metric used in BGP calculations. The combination of such indirect mechanisms create subtle dependencies. For instance, when OSPF link weights are changed to load balance the traffic in the network, inter-domain routing policy may be impacted. The outcome of the synthesis of indirect control mechanisms can be difficult to predict and exacerbates the complexity of network control [1].

The direct control paradigm avoids these problems because it forces the dependencies between control policies to become explicit. In direct control, a logically centralized entity called the decision element is responsible for creating all the state at every switch. As a result, any conflicts between the policy objectives can be detected at the time of state creation. With today's multiple independent and distributed mechanisms, these conflicts often only appear *in vivo* after some part of the configuration state has been changed by one of the mechanisms.

The direct control paradigm also simplifies the switch functionality. Because algorithms making control decisions are no longer run at switches, the only distributed functions to be implemented by switches are those that discover the neighborhood status at each switch and those that enable the control communications between the decision element and the switches. Thus, the switch software can be very light-weight. Yet, sophisticated control algorithms can be easily implemented with this minimal set of distributed functions.

The Tesseract (a tesseract is a 4-dimensional cube) system is based on the 4D architecture that advocates the decomposition of the network control plane into the *decision*, *dissemination*, *discovery*, and *data* planes. Tesseract implements two services to enable direct control:

Dissemination service: The dissemination service provides a logical connection between decision element and network switch nodes to facilitate direct control. The dissemination service only assumes network nodes are pre-configured with appropriate keys and can discover and communicate with direct physical neighbors. The dissemination service thus enables plug-and-play bootstrapping of the Tesseract system.

Node configuration service: The node configuration service provides an abstract packet lookup table interface that hides the details of the node hardware and software

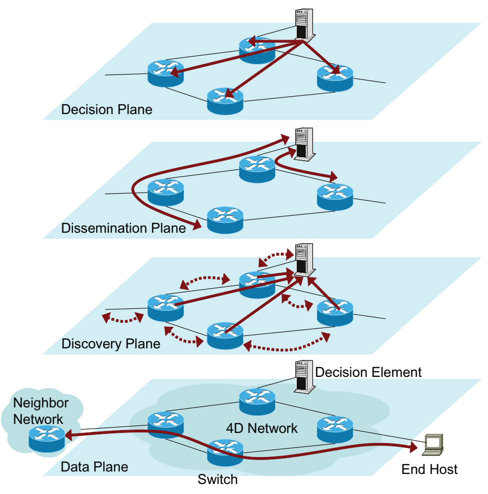


Figure 1: The 4D architectural concepts.

from the decision element. Each table entry contains a packet matching rule and the corresponding control actions. The decision element issues commands to the node configuration service through the logical connection provided by the dissemination service.

This paper presents the design, implementation, evaluation, and demonstration of the Tesseract system. To guide our design, we explicitly select a set of goals and devise solutions to address them. We deploy Tesseract on Emulab [2] to evaluate its performance. We show how Tesseract can rapidly react to link, node, and decision element failures and efficiently re-configure network switches in response. Also, micro-benchmark experiments show that the system can easily handle the intra-domain routing control for a thousand-node network. We then demonstrate Tesseract's flexibility by showing its applications in joint packet forwarding and policy based filtering in IP networks, and in link cost driven Ethernet packet forwarding.

2 From the 4D Architecture to Tesseract Design Goals

This section explains the key concepts in the 4D architecture. Since the 4D architecture describes a very large design space, we present the design goals we used to guide our design of the specific Tesseract system.

2.1 The 4D Architectural Concepts

The 4D architecture advocates decomposing the network control plane into four conceptual components: *decision*, *dissemination*, *discovery*, and *data* planes. These conceptual components are illustrated in Figure 1 and are explained below. For an in-depth discussion of the 4D architecture, please refer to [3].

Data plane: The data plane operates in network switches and provides user services such as IPv4, IPv6, or Ethernet packet forwarding. The actions of the data plane are based on the state in the switches, and this state is controlled solely by the decision plane. Example state in switches includes the forwarding table or forwarding information base (FIB), packet filters, flow-scheduling weights, queue-management parameters, tunnels and network address translation mappings, etc. The arrow in the figure represents an end-to-end data flow.

Discovery plane: Each switch is responsible for discovering its hardware capabilities (e.g., what interfaces are on this switch and what are their capacities? How many FIB entries can the switch hold?) and its physical connectivity to neighboring switches. A border switch adjacent to a neighboring network is also responsible for discovering the logical connectivity to remote switches that are reachable via that neighbor network (in today's environment, this may be implemented by an eBGP session). The dotted arrows in the figure represent the local communications used for discovering connectivity. The information discovered is then reported to the decision element in the decision plane via the logical connections maintained by the dissemination plane. The solid arrows in the figure represent these reporting activities. For backward compatibility, end hosts do not explicitly participate in the discovery plane.

Dissemination plane: The dissemination plane is responsible for maintaining robust logical channels that carry control information between the decision element and the network switches. The arrows in the figure represent the paths used by the logical channels. While control information may traverse the same set of physical links as the data packets in the data plane, the dissemination paths are maintained separately from the data paths so they can be operational without requiring configuration or successful establishment of paths in the data plane. In contrast, in today's networks, control and management information is carried over the data paths, which need to be established by routing protocols before use. This creates a circular dependency.

Decision plane: The decision plane consists of a logically centralized decision element that makes *all* decisions driving network control, such as reachability, load balancing, access control, and security. The decision element makes use of the information gathered by the discovery plane to make decisions, and these decisions are sent as commands to switches via the dissemination plane (shown as arrows in the figure). The decision element commands the switches using the node configuration service interface exposed by the network switches. While logically centralized as a single decision element, in practice multiple redundant decision elements may be used for resiliency.

2.2 Tesseract Design Goals

Tesseract is based on the general 4D architectural concepts, but these concepts admit a wide variety of design choices. We used the following goals to guide our decisions while designing Tesseract, and these goals can be roughly grouped into three categories. The first category concerns system performance and robustness objectives:

Timely reaction to network changes: Planned and unplanned network changes, such as switch maintenance and link failures, can cause traffic disruption. Tesseract should be optimized to react to network changes quickly and minimize traffic disruption.

Resilient to decision plane failure: Tesseract should provide built-in support for decision plane redundancy so that it can survive the failure of a decision element.

Robust and secure control channels: The logical channels for control communications maintained by Tesseract should continue to function in the presence of compromised switches, decision elements or failed links/nodes.

The next set of goals concern making Tesseract easy to deploy:

Minimal switch configuration: The Tesseract software on each switch should require no manual configuration prior to deployment except for security keys that identify the switch. We do, however, assume that the underlying switch allows Tesseract to discover the switch's properties at run-time.

Backward compatibility: Tesseract should require no changes to the end host software, hardware, or protocols. Thus, Tesseract can be deployed as the network control system transparently to the end users.

The final set of goals concerns making Tesseract a flexible platform:

Support diverse decision algorithms: Tesseract should provide a friendly platform on which diverse algorithms can be easily implemented to control networks.

Support multiple data planes: Tesseract should support heterogeneous data plane protocols (e.g., IP or Ethernet). Thus, the system should not assume particular data plane protocols and the dissemination service should be agnostic to the semantics of the control communications.

3 Design and Implementation of Tesseract

In this section, we present the design and implementation of Tesseract. We first provide an overview of the software architecture, and then discuss each component of the system in detail.

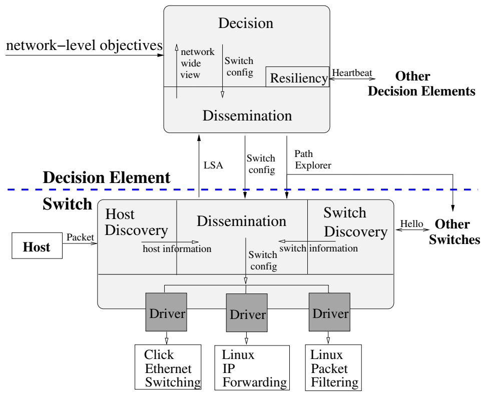


Figure 2: High-level overview of Tesseract.

3.1 System Overview

The Tesseract system is composed of two applications implemented on Linux. These applications are called the i h and the Decision Element ( ). Figure 2 illustrates the software organization of these applications.

The discovery plane implementation currently deals only with neighbor node discovery. It includes two modules, one for discovering hosts connected to the switch and the other for discovering other switches. The switch discovery module exchanges hello messages with neighbor switches to detect them, and creates Link State Advertisements (LSAs) that contain the status of its interfaces and the identities of the switches connected to the interfaces. The generated LSAs are reported to via the dissemination plane. To avoid requiring changes to hosts, the discovery plane identifies what hosts are connected to a switch by snooping the MAC and IP addresses on packets received on the interfaces that are not connected to another switch.

The dissemination plane is cooperatively implemented by both i h and . The dissemination service is realized by a distributed protocol that maintains robust logical communication channels between the switches and decision elements.

i h leverages existing packet forwarding and filtering components to implement the data plane. i h interacts with in the decision plane through the node configuration service interface. The interface is implemented by data plane drivers, which translate generic configuration commands from into specific configurations for the packet forwarding and filtering components.

implements the discovery, dissemination and decision planes. The discovery and dissemination plane functions are as outlined above. The decision plane constructs an abstract network model from the information reported by the switches and computes switch configuration commands for all the switches based on the specific

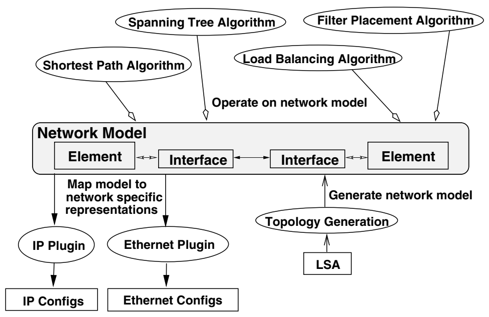


Figure 3: The network model separates general purpose algorithms from network specific mechanisms.

decision algorithm used. The computed switch configuration commands are sent to the switches via the dissemination service.

3.2 Decision Plane: Versatility, Efficiency and Survivability

The decision plane implements a platform for the deployment of network control algorithms. In addition, it implements mechanisms that enable the replication of the decision logic among multiple decision elements (DEs) so that DE failures can be tolerated.

Support diverse network control algorithms: In designing the decision plane, our focus is not to hard-wire sophisticated network decision logics into the system. Instead, our goal is to make the decision plane a friendly platform where any network control algorithm can be easily integrated and used to control any suitable network technology. Towards this end, we introduce an abstract network model to separate generic network control algorithms (e.g., shortest path computation, load balancing) from network specific mechanisms (e.g., IP, Ethernet).

Figure 3 illustrates the abstract network model. The model consists of node element and link interface objects, and is constructed from information discovered and reported by switches (e.g. LSA) through the dissemination service. Operating on this model, Tesseract currently implements four generic algorithms: incremental shortest path, spanning tree, joint packet filter/routing (Section 5.1), and link-cost-based traffic engineering (Section 5.2). Finally, technology-specific plug-ins translate the general control decisions into network specific configuration commands that are sent to switches via the dissemination service. These commands are then processed by the node configuration service at individual switches.

As an example, we implement an incremental shortest path algorithm [4] on the abstract network model, and the same code can be used to generate either IP routing table in IP networks or Ethernet forwarding entries in Ethernet.

Efficient network event processing: The DE must efficiently handle multiple simultaneous network changes, which the DE will receive as events communicated over the dissemination plane. We chose a different event processing architecture than that used in typical implementation of OSPF, where a hold-down timer is used to delay the start of route recomputation after an event arrives to force the batching of whatever events arrive during the hold-down window.

Instead, the Tesseract DE uses a *push timer*. The DE runs a decision thread that processes all queued events to update the network-wide view, starts the push timer as a deadline for pushing out new switch configuration commands, and then enters its computation cycle. After the compution of new forwarding state finishes, the DE will immediately push out the new commands if the push timer has expired, if the event queue is empty, or if the queued events do not change the network-wide view used in the computation. Otherwise, the DE will dequeue all pending events and re-compute.

We use a push timer instead of a fixed hold-down timer for two reasons. In the common case where a single link fails, the push timer avoid unnecessary waiting. The first LSA announcing the failure starts the route recomputation, and subsequent LSAs announcing the same failure do not change the network-wide view and so are ignored. In the less common case of multiple failures, a push timer may result in recomputation running more than once for the same event. However, since recomputation has latency on the same order as typical hold-down timers and DEs are unlikely to be CPU-limited, it is reasonable to trade off extra computation for faster reconvergence.

The DE also records the state that has been pushed to each switch and uses delta-encoding techniques to reduce the bandwidth required for sending configuration commands to the switches. Acknowledgments between DE and the node configuration service on each switch ensure the delta-encoded commands are received.

Provide decision plane resiliency: Our decision plane copes with DE failures using hot-standbys. At any time a single master DE takes responsibility for configuring the network switches, but multiple DEs can be connected to the network. Each standby DE receives the same information from the switches and performs the same computations as the master. However, the standby DEs do not send out the results of their computations.

The master DE is selected using a simple leader election protocol based on periodic DE heartbeats that carry totally ordered DE priorities. Each DE has a unique priority, and at boot time it begins flooding its priority with a heartbeat message every heartbeat period (e.g., 20 ms). Each DE listens for heartbeats from other DEs for at least five times the heartbeat period (we assume that 5 times heartbeat period will be greater than the maximum latency of a packet crossing the network). After this waiting period, the DE that has the highest priority among all received heartbeats decides to be the master and begins sending commands to switches. When the master DE receives a heartbeat from a DE with a higher priority than its own, it immediately changes into a standby DE and ceases sending commands to switches. A DE also periodically floods a path explorer message, which has the effect of triggering switches to reply with their current state. In this way, a new DE can gather the latest switch state. Switches simply process commands from any DE. Authentication is handled by the dissemination plane and is discussed next.

3.3 Dissemination Plane: Robustness and Security

The goal of the dissemination plane is to maintain robust and secure communication channels between each DE and the switches. With respect to robustness, the dissemination plane should remain operational under link and node failure scenarios. With respect to security, the network should remain operational when a switch or even a DE is compromised.

Observing that the traffic pattern in dissemination plane is few-to-many (switches communicate not with each other, but only with the DEs), we adopt an asymmetric design where the dissemination module at a DE node implements more functionality than the dissemination module at a switch.

Dissemination plane design overview: Tesseract's dissemination plane is implemented using source routes. Each control message is segmented into dissemination frames, and each frame carries in its header the identity of the source, destination, and the series of switches through which it must pass. We choose a source routing solution because: (1) It requires the minimal amount of routing state and functionality in each switch. Each switch needs only to maintain the routes to the DEs. (2) Source routes provide very flexible control over routing, as a different path can be specified for each destination, making it easy to take advantage of preferred paths suggested by the decision plane. (3) Combining source routing with the few-to-many communication pattern enable us to design a dissemination plane with desirable security properties, as discussed below. To protect control communications from user data traffic, the queuing of dissemination frames is separate from user data traffic and dissemination frames have higher transmission priority. To protect the source-routes from being misused by adversaries inside the network, we encrypt them at each hop before they are forwarded.

Threat model: Tesseract is designed to cope with the following threats: (1) Adversaries can compromise a switch, gaining full control over it including the ability to change the way dissemination packets are forwarded; (2) A compromised switch can piggyback data on packets to collude with other compromised switches downstream; (3) A compromised switch can peek into dissemination plane data to try to learn the network topology or location of critical resources; and (4) Adversaries can compromise a DE and use it to install bad forwarding state on the switches.

Bootstrapping security: The Tesseract trust model is based on a *network certificate* (i.e., a signed public key for the network) — all the other keys and certificates are derived from the network certificate and can be replaced while network continues operating. Switches will accept commands from any DE holding a DE certificate that is signed by the network certificate. The private key of the network certificate is secret-shared [5] among the DEs, so that any quorum of DEs can cooperate to generate a new DE certificate when needed.

When a switch is first deployed, the network certificate and a DE certificate are installed into it. This is done by plugging a USB key containing the certificates into each switch or as part of the default factory configuration of the switch before it is deployed in the field. The switch then constructs a DeviceID, which can be as simple as a randomly generated 128-bit number, and a private/public key pair. The switch stores the network and DE certificates, its DeviceID, and its key pair into nonvolatile memory. The switch then encrypts the information with the public key of the DE, and writes it back onto the USB key. When the USB key is eventually inserted into a DE, the DE will have a secret channel to each device and a list of the valid DeviceIDs. As each switch communicates with a DE for the first time, it uses ISAKMP [6] and its private/public keys to establish a shared-secret key known only by that switch and the DE. All subsequent dissemination plane operations use symmetric cryptography.

Computing dissemination plane routes: Dissemination plane routes are computed by each decision element flooding a path explorer message through the network. To ensure fast recovery from link failures, the path explorer is sent periodically every 20 ms in our prototype, and can be triggered by topology updates.

Onion-encryption (or encapsulated encryption) is used in path explorers to support dissemination security. The DE initiates the path explorer by embedding its DeviceID as the source route and flooding it over all its ports. When a switch receives the path explorer, it (1) optionally verifies the route to the DE contained in the path explorer; (2) records the source route; (3) encrypts the existing source route using the secret key it shares with the DE that sent the path explorer; (4) appends its own DeviceID to the path explorer in plain text; and (5) floods the path explorer out its other interfaces. Path explorers carry sequence numbers so that switches can avoid unnecessary re-flooding.

To send data to a DE, a switch uses the encrypted source route it recorded from a path explorer sent by that DE. When an upstream switch receives the message, it decrypts the source-route using its secret key. This reveals the ID of the next hop switch along the path to the DE. By successive decryption of the source route by the on-route switches, dissemination plane packets are delivered to the DE. Since the DE knows the secret-key of every switch, it can construct an onion-encrypted route to any switch it desires.

As part of the negotiation of its secret key over ISAKMP, each switch learns whether it is required to perform the optional source route verification in step (1) before forwarding a path explorer. If verification is required, the switch first checks a cache of source routes from that DE to see if the source route has already been verified. If the source route is not known to be valid, the switch forwards the source route to the DE in a signed VERIFY packet. Since the DE knows the secret keys of all the switches, it can iteratively decrypt the source route and verify that each hop corresponds to link it has learned about in an LSA. Once verified, the DE sends a VERIFYOK message to the switch using the extracted source route, confirming the validity of the route. The DE confirmation is signed with a HMAC computed using the secret key of the destination switch to prevent it from being tampered or forged.

Security properties: The optional verification step exposes a classic trade-off between security and performance. In Tesseract, we provide a dissemination plane with two different levels of security. The network operator can choose the semantics desired.

The basic security property is that a compromised switch cannot order other switches to install invalid forwarding state or forge LSAs from other switches. This is achieved by each switch having a secret key shared only with the DE.

If path explorers are *not* verified before being forwarded, a compromised switch can forge path explorers that artificially shorten its distance to the DE and attract dissemination plane traffic from other switches (e.g., so the attacker can drop or delay the traffic). Compromised switches can also communicate with each other over the dissemination plane to coordinate attacks.

If path explorers *are* verified before being forwarded, a compromised switch cannot lie about its distance to the DE. Also, compromised switches are prevented from communicating arbitrarily over the dissemination plane unless they are directly connected. This is because the DE will not validate a source route that originates and ends at switches. A switch also cannot discover the identity or connectivity of another switch that is two or more hops away. This prevents attackers from identifying and targeting critical resources in the network.

The cost of the extra security benefits provided by verifying source routes is the extra latency during reconvergence of the dissemination plane. If a link breaks and a switch receives path explorers over a source route it has not previously verified, it must wait a round-trip time for the verification to succeed before the switches downstream can learn of the new route to the DE. One approach to minimize this penalty is for the DE to prepopulate the verified source route tables of switches with the routes that are most likely to be use in failure scenarios. A triggered path explorer flooded by the DE in response to link failure will then quickly inform each switch which preverified routes are currently working.

Surviving DE compromise: As a logically centralized system, if a DE were compromised, it could order switches to install bad forwarding state and wreck havoc on the data plane. However, recovery is still possible. Other DEs can query the forwarding state installed at each switch and compare it to the forwarding state they would have installed, allowing a compromised or misbehaving DE to be identified. Because the private key of the network certificate is secret-shared, as long as a quorum of DEs remain uncompromised they can generate a new DE certificate and use the dissemination plane to remotely re-key the switches with this new DE certificate.

Notice that while a compromised DE can totally disrupt data plane traffic, it *cannot* disrupt the dissemination traffic between other DEs and the switches. This is one of the benefits of having control traffic traversing a secured dissemination plane that is logically separate from paths traversed by data packets. Once re-keyed, the switches will ignore the compromised DEs.

As a point of comparison, in today's data networks recovering from the compromise of a management station is hard as the compromised station can block the uncompromised ones from reaching the switches. At the level of the control plane, the security of OSPF today is based on a single secret key stored in plain-text in the configuration file. If any switch is compromised, the key is compromised, and incorrect LSAs can be flooded through the network. The attacker could then DoS all the switches by forcing them to continuously rerun shortest path computation or draw traffic to itself by forging LSAs. Since a distributed link-state computation depends on all-to-all communications among the switches, one alternative to using a single shared key is for each switch to negotiate a secret key with every other switch. Establishing this $O(n^2)$ mesh of keys requires every switch to know the public key of every other switch. Both key establishment and revocation are more complex when compared to the direct control paradigm of Tesseract.

3.4 Discovery Plane: Minimizing Manual Configurations

The discovery plane supports three categories of activities: (1) providing the DE with information on the state of the network; (2) interacting with external networks and informing the DE of the external world; and (3) bootstrapping end hosts into the network.

Gathering local information: Since misconfiguration is the source of many network outages, the 4D architecture eliminates as much manually configured state as possible. In the long term vision, the switch hardware should self-describe its capabilities and provide run-time information such as traffic load to the discovery plane. The current Tesseract implementation supports the discovery of physical switch neighbors via periodic HELLO message exchanges. Switches are identified by the same DeviceID used in the dissemination plane.

Interacting with external networks: The DE directs the border switches that peer with neighbor networks to begin eBGP sessions with the neighbor switches. Through this peering, the DE discovers the destinations available via the external networks. Rather than processing the BGP updates at the switches, the switches simply report them to the DE via the dissemination service, and the DE implements the decision logic for external route selection. The DE sends the appropriate eBGP replies to the border switches, as well as configuring external routes directly into all the switches via the dissemination service. RCP [7] has already demonstrated that the overall approach of centralized BGP computation is feasible, although they continue to use iBGP for backward compatibility with existing routers.

It is important to note that an internal link or switch failure in a Tesseract network does not lead to massive updates of external routes being transmitted from the DE to the switches. The reason is that external routes identify only the egress points. External and internal routes are maintained in two separate tables and are combined locally at switches to generate the full routing table. This is identical to how OSPF and BGP computed routes are combined today. In general, an internal link or switch failure does not change external routes and thus no update to them is necessary.

Bootstrapping end hosts: For backward compatibility, end hosts do not directly participate in Tesseract discovery plane.

In networks running IP, the discovery plane acts as a DHCP proxy. The DE configures each switch to tunnel DHCP requests to it via the dissemination service. Whenever a host transmits a DHCP request, the DE learns the MAC address and the connection point of the host in the network. The DE can then assign the appropriate IP address and other configuration to the host.

In networks operating as a switched Ethernet LAN, the discovery plane of a switch reports the MAC address and the connection point of a newly appeared end host to the DE. The DE then configures the network switches appropriately to support the new host. Section 5.2 describes how we use Tesseract to control a switched Ethernet LAN and provide enhancements.

3.5 Data Plane: Support Heterogeneity

The data plane is configured by the decision plane via the node configuration service exposed by the switches. Tesseract abstracts the state in the data plane of a switch as a lookup table. The lookup table abstraction is quite general and can support multiple technologies such as the forwarding of IPv4, IPv6, or Ethernet packets, or the tunneling and filtering of packets, etc.

Tesseract's data plane is implemented using existing Linux kernel and Click components. For each component, we provide a driver to interface the component with the Tesseract decision plane as shown in Figure 2. The drivers model the components as lookup tables and expose a simple `ri    a` interface to provide the node configuration service to the DE. For example, when the DE decides to add or delete an IP routing or Ethernet forwarding table entry, it sends a `a  _a    _   r` or `     _a   _   r` command through the `ri    a` interface, and the driver is responsible for translating the command into component-specific configurations. This allows the algorithms plugged into the DE to implement network control logic without dealing with the details of each data-plane component. We implemented three drivers and describe their details next.

Linux IP forwarding kernel: The Linux kernel can forward packets received from one network interface to another. To determine the outgoing network interface, the Linux kernel uses two data structures: a Forwarding Information Base (FIB) that stores all routes, and a routing cache that speeds up route search. As in all Tesseract data plane drivers, the driver for Linux IP forwarding kernel implements the `ri    a` interface. The driver interprets commands from the DE, creates a `r     r` structure with the route to add or delete, and invokes the `i` system call to modify the FIB. We set `r          i    r     mi _   a` to zero so that the routing cache is flushed immediately after the FIB is modified.

Click router: We use Click for forwarding Ethernet frames. The driver for Click includes two parts: an implementation of the `ri    a` interface, and a Click element package called the `    i   h` that is integrated into Click. The implementation of `ri    a` parses commands and executes those commands by exchanging control messages with the 4DSwitch element in the Click process via a TCP channel. The i h element maintains an Ethernet forwarding table and updates the table according to the received control messages. To control the data forwarding behavior of Click, the i h element overrides the Click m h function and directs incoming traffic to the outgoing port(s) specified in the i h forwarding table.

netfilter/iptables: Tesseract uses netfilter/iptables to implement reachability control in IP networks. The driver for netfilter/iptables translates commands into iptables rules (e.g., i h j) and forks an iptables process to install the rules.

3.6 Decision/Dissemination Interface

In designing the interface between the decision plane and the dissemination plane, there is a tension between the conflicting goals of creating a clean abstraction with rigid separation of functionality and the goal of achieving high performance with the cooperation of the decision and dissemination planes.

The key consideration is that the dissemination plane must be able to function independently of the decision plane. Our solution is to build into the dissemination plane a completely self-contained mechanism for maintaining connectivity. This makes the dissemination plane API very simple, giving the basic decision plane only three interface functions: , , which sends control information to a specific switch, , which floods control information to all switches, and i r a , which identifies the decision plane function that handles incoming information.

However, to optimize the performance of the dissemination plane, we add two interface functions: i k ai r i k , which the DE uses to identify a known failed link to the dissemination plane so the dissemination plane can avoid it immediately, and r rr , r , which the DE uses to suggest a specific source route for carrying control information to switch . This solution enables a sophisticated DE to optimize the dissemination plane to its liking, but also allows the simplest DE to fully function.

4 Performance Evaluation

In this section, we evaluate Tesseract to answer the following questions: How fast does a Tesseract-controlled network converge upon various network failures? How large a network can Tesseract scale to and what are the bottlenecks? How resilient is Tesseract in the presence of decision-element failures?

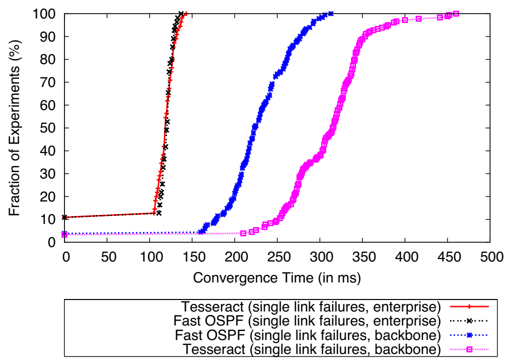


Figure 4: CDF of convergence times for single link failures in enterprise and backbone networks. We pick one link to fail at a time and we enumerate all the links to get the distribution of convergence times. The zero convergence times are caused by failures disconnecting switches at the edge of the network.

4.1 Methodology

We perform both emulation and simulation experiments. We use Emulab to conduct intra-domain routing experiments using two different topologies. The first topology is an ISP backbone network (AS 3967) from Rocketfuel [8] data that spans Japan, U.S., and Europe, with a maximum round trip delay of 250 ms. The other is a typical enterprise network with negligible propagation delay from our earlier study [9].

Emulab PCs have 4 interfaces each, so routers that have more than 4 interfaces are modeled by chaining together PCs to create a “supernode” (e.g., a router with 8 interfaces will be represented by a string of 3 Emulab PCs). As a result, the backbone network is emulated by 114 PCs with 190 links, and the enterprise network is emulated by 40 PCs with 60 links. For each Tesseract experiment, there are 5 decision elements — these run on “pc3000” machines that have a 3GHZ CPU and 2GB of RAM. To inject a link failure, we bring down the interface with the i i command. To inject a switch failure, we abruptly terminate all the relevant software running on a switch.

So that we evaluate the worst-case behavior of the control plane, we measure the time required for the *entire* network to reconverge after an event. We calculate this network convergence time as the elapsed time between the event occurring and the last forwarding state update being applied at the last switch to update. We use Emulab’s NTP (Network Time Protocol) servers to synchronize the clocks of all the nodes to within 1 millisecond.

As a point for comparison, we present the performance of an *aggressively tuned* OSPF control plane called Fast OSPF. Fast OSPF's convergence time represents the best possible performance achievable by OSPF and it is determined by the time to detect a link failure and the one way propagation delay required for the LSA flood. Such uniform and aggressive tuning might not be practical in a real network as it could lead to CPU overload on older routers, but Fast OSPF serves as a useful benchmark.

We implemented Fast OSPF by modifying Quagga 0.99.4 [10] to support millisecond timer intervals. There are four relevant timers in Quagga: (1) the hello timer that sets the frequency of HELLO messages; (2) the dead timer that sets how long after the last HELLO is received is the link declared dead; (3) the delay timer that sets the minimum delay between receiving an LSA update and beginning routing computation; and (4) the hold-down timer that sets the minimum interval between successive routing computations. For Fast OSPF, we use hello timer = 20 ms, dead timer = 100 ms, delay timer = 10 ms (to ensure a received LSA is flooded before routing computation begins), and 0 ms for the hold-down timer. Tesseract uses the same hello and dead timer values to make direct comparison possible. There is no need for the delay timer or the hold-down timer in Tesseract.

4.2 Routing Convergence

Common concerns with using a logically centralized DE to provide direct control are that reconvergence time will suffer or the DE will attempt to control the network using an out-of-date view of the network. To evaluate these issues, we measure intra-domain routing convergence after single link failures, single switch failures, regional failures (i.e., simultaneous multiple switch failures in a geographic region), and single link flapping.

Single link failures: Figure 4 shows the cumulative distribution of convergence times of Tesseract and Fast OSPF for all single link failures in both topologies (Some convergence times are 0 because the link failure partitioned a stub switch and no forwarding state updates were required). First, consider the enterprise network scenario where the network propagation delay is negligible. For Fast OSPF, which represents an ideal target for convergence time, its performance is primarily a function of the link failure detection time, which is controlled by the dead timer value (100 ms), and the time to compute and install new routes. Even though Tesseract has a single DE machine compute all the routes, its performance is nearly identical to that of Fast OSPF, thanks to the usage of an efficient dynamic shortest path algorithm and the delta encoding of switch configurations. The only observable difference is that Tesseract's convergence time has a slightly larger variance due to the variability of the dynamic shortest path algorithm on different failed links.

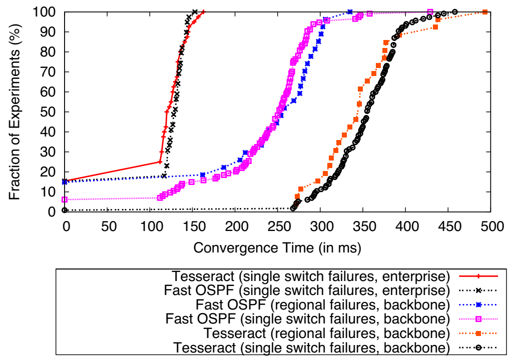


Figure 5: CDF of convergence times for single switch failures and regional failures.

In the backbone network scenario, propagation delay becomes an important factor as switch-to-switch RTT ranges from 1 ms to 250 ms. Tesseract's convergence requires the link state update to be transmitted to the DE and the new switch configurations to be transmitted back to the switches. On the other hand, Fast OSPF only requires the one-way flooding of the link state update. This is why Tesseract's convergence time is roughly a one-way delay slower than Fast OSPF. In return, however, the direct control paradigm enabled by Tesseract allows other control functions, such as packet filtering, to be implemented together with intra-domain routing in a simple and consistent manner.

Switch failures and regional failures: Next, we examine the convergence time under single switch failures and regional failures. To emulate regional failures, we divide the backbone topology into 27 geographic regions with each region containing a mean of 7 and a maximum of 26 switches, and we simultaneously fail all switches in a region.

Figure 5 compares the cumulative distributions of convergence times of Tesseract and Fast OSPF on switch and regional failures. In the enterprise network, again, the performance of Tesseract is very similar to that of Fast OSPF. In the backbone network, the difference between Tesseract and Fast OSPF is still dominated by network delay, and both are able to gracefully handle bursts of network state changes. There are two additional points to make. First, Fast OSPF has more cases where the convergence time is zero. This is because the 10 ms delay timer in Fast OSPF is acting as a hold-down timer. As a result, Fast OSPF does not react immediately to individual link state updates for a completely failed switch and sometimes that can avoid unnecessary configuration changes. In Tesseract, there is no hold-down timer, so it reacts to some link state updates that are ultimately inconsequential. Second, in some cases, Tesseract has faster convergence time in regional failure than in single

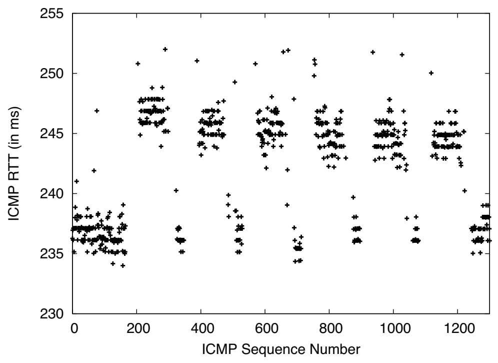


Figure 6: Effects of link flapping on ICMP packets sent at a rate of 100 packets/sec.

switch failure. The reason is that the large number of failed switches in regional failure reduces the amount of configuration updates Tesseract needs to send.

Link flapping: From the earliest days of routing in the Internet there has been concern that a rapidly flapping link could overload the control plane and cause a widespread outage worse than the failure of that single link. Using Emulab we conduct an experiment to show the effects of link flapping on the end-to-end behavior of Tesseract. On the emulated backbone network, we i the Tokyo node from the Amsterdam node at an interval of 10 ms and measure the RTT. We start to flap the link between Santa Clara and Herndon 2 seconds into the experiment. The flapping link is up for 100 ms and then down for 2 seconds. As the link flaps, the route from Tokyo to Amsterdam oscillates between a 10-hop path traversing Santa Clara, Herndon, Weehawken, and London with an average RTT of 240 ms, and a 12-hop path through San Jose and Oak Brook with an average RTT of 246 ms, as shown in Figure 6.

This experiment demonstrates that a logically centralized system like Tesseract can handle continual network changes. It is also worth mentioning that the Tesseract decision plane makes it easy to plug in damping algorithms to handle this situation in a more intelligent way.

4.3 Scaling Properties

Another concern with a logically centralized system like Tesseract is can it scale to size of today's networks, which often contain more than 1,000 switches. Since Emulab experiments are limited in size to at most a few hundred switches, we perform several simulation experiments to evaluate Tesseract's scaling properties. This evaluation uses a DE running the same code and hardware as the previous evaluations, but its dissemination plane is connected to another machine that simulates the control plane of the network.

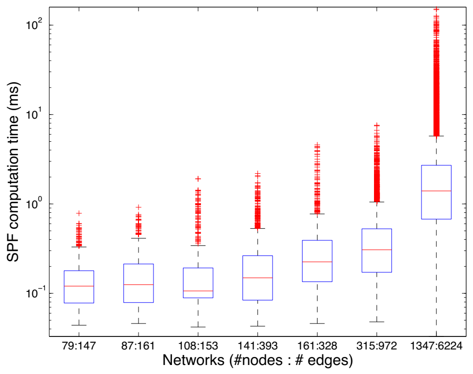


Figure 7: CPU time for computing incremental shortest paths for various Rocketfuel topologies in logarithmic scale. The box shows the lower quartile, upper quartile and median. The whiskers show the min and max data values, out to 1.5 times the interquartile range, and outliers are plotted as '+'s.

We evaluate Tesseract's scalability on a set of Rocketfuel topologies with varying sizes. For each topology, we independently fail each link in the graph and measure the time for the DE to compute new forwarding state and the size of the state updates.

DE Computation Time: Every time a failure occurs in the network, the decision element needs to recompute the forwarding tables for the switches based on the new state of the network. Figure 7 shows the results of DE path computation time. As shown in the figure, even in the largest network of 1347 nodes and 6244 edges, the worst case recomputation time is 151 ms and the 99th percentile is 40 ms.

Bandwidth Overhead of Control Packets: Each time the DE computes new forwarding state for a switch, it needs to push out the new state to the switch. Figure 8 plots the number of control bytes that the DE pushes out for independent link failures with different topologies. As shown in the figure, the worst case bandwidth overhead is 4.4MB in the largest network of 1347 nodes and 6244 edges. This is a scenario where 90% of the switches must be updated with new state.

Notice that the bandwidth overhead reported here includes only intra-domain routes. Even when a Tesseract network carries external BGP routes, the amount of forwarding state expected to change in response to an internal link failure will be roughly the same. Switches use two-level routing tables, so even if default-free BGP routing tables are in use, the BGP routes only change when the egress point for traffic changes — not when internal links fail. As has been pointed out by many [11, 7], Internet routing stability would improve if networks did not change egress points solely because the local cost changed, and Tesseract's framework for direct control makes it easier to implement this logic.

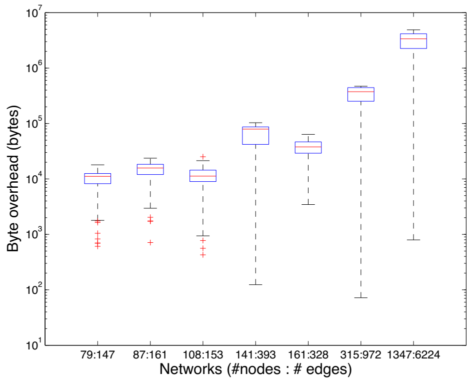


Figure 8: Switch configuration traffic sent out on a single link failure for various Rocketfuel topologies in logarithmic scale.

	Min	Mean	Max	SD
Backup DE takes over	130 ms	142 ms	155 ms	6 ms

Table 1: Minimum, mean, and maximum times, and standard deviation for DE failover in DE failure experiments on the backbone network.

4.4 Response to DE Failure and Partition

This section evaluates decision plane resiliency by measuring the *DE failover time*, defined as the time from when the master DE is partitioned to when a standby DE takes over and becomes the new master DE. We use the backbone network topology and perform 10 experiments in which the master and stand-by DEs are 50 ms apart.

DE failure: Failure of any DE but the master DE is harmless, since in Tesseract the other DEs are hot standbys. To evaluate the effect of the failure of the master DE, we abruptly shutdown the master DE. Table 1 shows the time required for a new DE to take control of the network after the master DE fails. As expected, the average failover time is approximately 140 ms, which can be derived from a simple equation that describes the expected failover time: ($DEDeadTime + PropagationDelay - HeartbeatInterval/2 = 100ms + 50ms - 10ms$).

Network partition: We inject a large number of link failures into the backbone topology to create scenarios with multiple network partitions. In the partition with the original master DE, Tesseract responds in essentially the same manner as in the regional-failure scenarios examined in Section 4.2, since the original master DE sees the partition as a large number of link failures. In the partitions that do not contain the original master, the convergence time is the same as when the master DE fails.

Just as network designers can choose to build a topology that has the right level of resistance against network partition (e.g., a ring versus a complete graph), the designers can intelligently select locations for placing redundant DEs to defend against network partition.

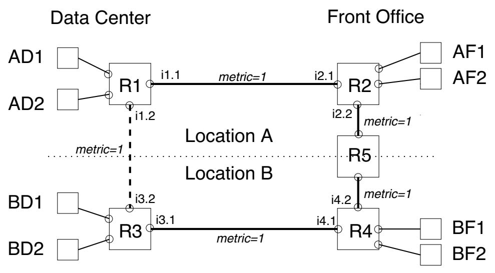


Figure 9: Enterprise network with two locations, each location with a front office and a data center. The dashed link is added as an upgrade.

5 Tesseract Applications

In this section, we demonstrate two applications that take advantage of Tesseract's direct control paradigm.

5.1 Joint Control of Routing and Filtering

Today, many enterprise networks configure packet filters to control which hosts and services can reach each other [9]. Unfortunately, errors in creating network configurations are rampant. The majority of disruptions in network services can be traced to mis-configurations [12, 13]. The situation with packet filters is especially painful, as routes are automatically updated by routing protocols to accommodate topology changes, while there is no mechanism to automatically adapt packet filter configurations.

The Tesseract approach makes joint routing and filtering easy. The decision logic takes as input a specification of the desired security policy, which lists the pairs of source and destination subnets that should or should not be allowed to exchange packets. Then, in addition to computing routes, for each source-destination subnet pair that is prohibited from communicating, the DE initially places a packet filter to drop that traffic on the interface closest to the destination. The decision logic then further optimizes filter placement by pulling the filters towards the source of forbidden traffic and combining them until further pulling would require duplicating the filters.

As a concrete example, consider the network in Figure 9. This company's network is spread across two locations, A and B. Each location has a number of front office computers used by sales agents (AF1-2 and BF1-2) and a data center where servers are kept (AD1-2 and BD1-2). Initially, the two locations are connected by a link between the front office routers, R2 and R4, over which inter-office communications flow. The routing metric for each link is shown in italics. Later, a dedicated link between the data centers (shown as a dashed line between R1 and R3) is added so the data centers can use each other as remote backup locations. The security policy is that front-office computers can communicate with the other location's front office computers and with the local data center's servers, but not the data center of the other location. Such policies are common in industries like insurance, where the sales agents of each location are effectively competing against each other.

We experimentally compared the Tesseract-based solution with a conventional solution that uses OSPF and manually placed packet filters. During the experiments we generate data traffic from AF1 to BF1 (which should be permitted) and from AF1 to BD1 (which should be forbidden) at 240 packets per second and monitor for any leaked or lost packets. In the OSPF network, the filter is manually placed on interface i3.1 to prevent A's front office traffic from reaching BD. After allowing the routing to stabilize, we add a new link between the data centers (dotted line in Figure 9). In the OSPF network, OSPF responds to the additional link by recomputing routes and redirects traffic from AF to BD over the new link, bypassing the packet filter on interface i3.1 and creating a security hole that will have to be patched by a human operator. In contrast, Tesseract computes both new routes *and new packet filter placements appropriate for those routes* and loads into the routers simultaneously, so no forbidden traffic is leaked. Most importantly, once the security policy is specified, it is automatically enforced with no human involvement required.

5.2 Link Cost Driven Ethernet Switching

Ethernet is a compelling layer-2 technology: large switched Ethernets are often used in enterprise, data center, and access networks. Its key features are: (1) a widely implemented frame format; (2) support for broadcasting frames, which makes writing LAN services like ARP, and DHCP significantly easier; and (3) its transparent address learning model, which means hosts can simply plug-and-play. Unfortunately, today's Ethernet control plane is primitive [14, 15, 16]. Based on routing frames along a spanning tree of the switches, it makes very inefficient use of the available links. Convergence time in response to failures can be long, as the IEEE 802.1D Rapid Spanning Tree Protocol (RSTP) is known to count to infinity in common topologies.

We have implemented a Tesseract control plane for Ethernet that preserves all three beneficial properties, avoids the pitfalls of a distributed spanning tree protocol, and improves performance. The DE first creates a spanning tree from the discovered network topology and generate default forwarding entries for the switches

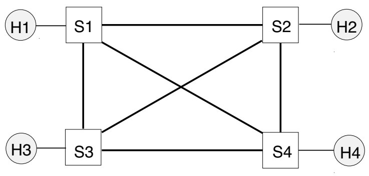


Figure 10: Full-mesh Ethernet topology.

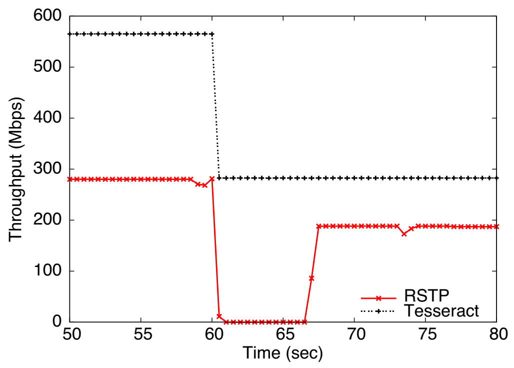


Figure 11: Aggregate network throughput, RSTP versus Tesseract. S1 fails at 60 second.

that follow the tree — this enables traditional tree-based broadcast. Additionally, when an end host sends its first frame to its first-hop switch, the switch notifies the DE of the newly discovered end host via the dissemination service. The DE then computes appropriate paths from all switches to that end host and adds the generated forwarding entries to the switches. From then on, all frames destined to the end host can be forwarded using the specific paths (e.g., shortest paths) instead of the spanning tree.

To experimentally illustrate the benefits of the Tesseract approach, we use the topology shown in Figure 10 on Emulab. The four switches are connected by 100 Mbps Ethernet links, and each end host is connected to one switch via a 1 Gbps Ethernet link. We run i r [17] TCP servers on the four end hosts and simultaneously start six TCP flows. They are H1 to H2, H1 to H3, H1 to H4, H2 to H3, H2 to H4, and H3 to H4. In the first experiment, the network is controlled by Tesseract using shortest path as the routing policy. In the second experiment, the network is controlled by an implementation of IEEE 802.1D RSTP on Click.

Figure 11 shows the aggregated throughput of the network for both experiments. With the Tesseract control plane, all six TCP flows are routed along the shortest paths, and the aggregate throughput is 570 Mbps. At time 60 s, switch S1 fails and H1 is cut off. The Tesseract system reacts quickly and the aggregate throughput of the remaining 3 TCP flows stabilizes at 280 Mbps. In contrast, in a conventional RSTP Ethernet control plane,

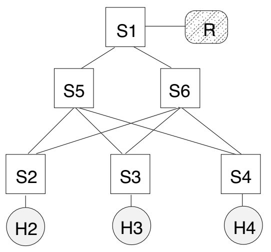


Figure 12: Typical Ethernet topology gadget.

forwarding is performed over a spanning tree with S1 as the root. This means the capacities of the S2-S3, S2-S4, and S3-S4 links are totally unused. As a result, the aggregate throughput of the RSTP controlled network is only 280 Mbps, a factor of two less than Tesseract. When switch S1 fails at time 60 s, RSTP tries to reconfigure the spanning tree to use S2 as the root and begins a count-to-infinity. The combination of frame loss when ports oscillate between forwarding/blocking state and TCP congestion control back-off means the throughput does not recover for many seconds. When RSTP has finally reconverged, the aggregate throughput is again substantially less than the Tesseract network.

As a second example of the value of being able to change the decision logic and the ease with which Tesseract makes this possible, consider Figure 12. This topology gadget is a typical building block found in Ethernet campus networks [18] which provides protection against any single link failure. Basic Ethernet cannot take advantage of the capacities of the redundant links since RSTP forms a spanning tree with S1 as the root, and the S2-S6, S3-S6, and S4-S6 links only provide backup paths and are not used for data forwarding. As a result, traffic flows from H2, H3, and H4 to R must share the capacity of link S1-S5. In contrast, when there exists two or more equal cost paths from a source to a destination, the Tesseract decision logic breaks the tie by randomly picking a path. By centralizing the route computations and using even such simple load-balancing heuristics, Tesseract is able to take advantage of the multiple paths and achieve a substantial increase in performance. In our example, the capacities of both link S1-S5 and S1-S6 are fully utilized for a factor of two improvement in aggregate throughput over RSTP.

The examples in this section illustrate the benefit of the direct control paradigm, where the only distributed functions to be implemented by network switches are those that discover the neighborhood status at each switch and those that enable the control communications between the DE and the switches. As a result, it becomes easy to design and change the decision logics that control the network. There is no need to design distributed protocols that attempt to achieve the desired control policies.

6 Related Work

Previous position papers [19, 3] have laid down the conceptual framework of 4D, and this paper provides the details of an implementation and measured performance. The Routing Control Platform (RCP) [7, 20] and the Secure Architecture for the Networked Enterprise (SANE) [21] are the most notable examples that share conceptual elements with 4D.

RCP is a solution for controlling inter-domain routing in IP networks. RCP computes the BGP routes for an Autonomous Systems (AS) at centralized servers to give the operators of transit networks more control over how BGP routing decisions are made. The RCP servers have a very similar role as the decision plane in 4D. For backward compatibility, the RCP servers use iBGP to communicate with routers. This iBGP communication channel has a role similar to the dissemination plane in 4D.

SANE is a solution for enforcing security policies in an enterprise network. In a SANE network, communications between hosts are disabled unless they are explicitly allowed by the domain controller. Switches only forward packets that have authentic secure source routes attached to them. For communications between switches and the domain controller, SANE constructs a spanning tree rooted at the domain controller. This spanning tree has a role similar to the dissemination plane in Tesseract.

Tempest [22] proposes an alternate framework for network control, where each switch is divided into switchlets and the functionality of each switch is exposed through a common interface called Ariel. Tempest allows multiple control planes to operate independently, each controlling their own virtual network composed of the switchlets, and the framework has been used on both MPLS and ATM data planes. Tesseract's dissemination plane provides a complete bootstrap solution, where Tempest's implementation assumed a pre-existing IP-over-ATM network for communication with remote switches. While both projects abstract switch functionality, Tesseract does not assume that switches can be fully virtualized into independent switchlets, and it leaves resource allocation to the decision logic.

FIRE [23] presents a framework to ease the implementation of distributed routing protocols by providing a secure flooding mechanism for link-state data, hooks to which route computation algorithms can be attached, and a separate FIB used for downloading code into the router. Tesseract eases the implementation of centralized network control algorithms by assembling a network-wide view, enabling direct control via a robust and self-bootstrapping dissemination plane, and providing redundancy through the election of DEs.

7 Summary

This paper presents the design and implementation of Tesseract, a network control plane that enables direct control. In designing Tesseract, we paid particular attention to the robustness of the decision plane and the dissemination plane. The security of Tesseract is enhanced by the mechanisms built into the dissemination service. The system is designed to be easily reusable and we demonstrated how Tesseract can be used to control both Ethernet and IP services. Finally, good performance is achieved by adopting efficient algorithms like incremental shortest path and delta encoding of switch configuration updates.

We find that Tesseract is sufficiently scalable to control intra-domain routing in networks of more than one thousand switches, and its reconvergence time after a failure is detected is on the order of one round-trip propagation delay across the network.

The most important benefit of Tesseract is that it enables direct control. Direct control means sophisticated control policies can be implemented in a centralized fashion, which may be much easier to understand and deploy than a distributed protocol. Direct control also means the software running on each switch is simplified, with potential benefits for operators and vendors. We strongly believe that the direct control paradigm is the right approach in the long run, as there is a clear trend towards ever more sophisticated network control policies.

Acknowledgments

Andy Myers contributed greatly to the early stages of this project, and we would like to thank the anonymous reviewers and our shepherd Jon Crowcroft for helpful feedback on earlier versions of the paper. This research was sponsored by the NSF under ITR Awards ANI-0085920, ANI-0331653, and NeTS Grant CNS-0520187. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of Microsoft, NSF, or the U.S. government.

References

[1] A. Shaikh and A. Greenberg, "Operations and management of IP networks: What researchers should know," August 2005. ACM SIGCOMM Tutorial.

[2] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. Operating Systems Design and Implementation*, pp. 255–270, December 2002.

[3] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang, "A clean slate 4D approach to network control and management," *ACM Computer Communication Review*, October 2005.

[4] C. Demetrescu and G. F. Italiano, "A new approach to dynamic all pairs shortest paths," *J. ACM*, vol. 51, no. 6, pp. 968–992, 2004.

[5] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 1, pp. 612–613, 1979.

[6] D. Maughan, M. Schertler, M. Schneider, and J. Turner, "Internet Security Association and Key Management Protocol (ISAKMP)." RFC 2048, November 1998.

[7] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proc. ACM SIGCOMM Workshop on Future Directions in Network Architecture*, August 2004.

[8] N. Spring, R. Mahajan, and D. Wetheral, "Measuring ISP topologies with RocketFuel," in *Proc. ACM SIGCOMM*, August 2002.

[9] D. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjalmtysson, and A. Greenberg, "Routing design in operational networks: A look from the inside," in *Proc. ACM SIGCOMM*, August 2004.

[10] "Quagga Software Routing Suite."
h a a .

[11] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford, "Dynamics of hot-potato routing in IP networks," in *Proc. ACM SIGMETRICS*, June 2004.

[12] Z. Kerravala, "Configuration management delivers business resiliency." The Yankee Group, Nov 2002.

[13] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why do internet services fail, and what can be done about it," in *Proc. USENIX Symposium on Internet Technologies and Systems*, 2003.

[14] A. Myers, T. S. E. Ng, and H. Zhang, "Rethinking the service model: Scaling Ethernet to a million nodes," in *Proc. HotNets*, November 2004.

[15] R. Perlman, "Rbridges: Transparent routing," in *Proc. IEEE INFOCOM*, March 2004.

[16] K. Elmeleegy, A. L. Cox, and T. S. E. Ng, "On count-to-infinity induced forwarding loops in ethernet networks," in *Proc. IEEE INFOCOM*, 2006.

[17] "Iperf – The TCP/UDP Bandwidth Measurement Tool."
h a ar rj r .

[18] "Gigabit campus network design – principles and architecture." Cisco White Paper.

[19] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang, "Network-wide decision making: Toward a wafer-thin control plane," in *Proc. HotNets*, pp. 59–64, November 2004.

[20] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and Jacobus van der Merwe, "Design and implementation of a Routing Control Platform," in *Proc. NSDI*, May 2005.

[21] M. Casado, T. Garfinkel, A. Akella, M. Freedman, D. Boneh, N. McKeown, and S. Shenker, "SANE: A protection architecture for enterprise networks," in *Usenix Security*, August 2006.

[22] S. Rooney, J. van der Merwe, S. Crosby, and I. Leslie, "The Tempest: a framework for safe, resource assured, programmable networks," *IEEE Communications Magazine*, vol. 36, pp. 42–53, Oct 1998.

[23] C. Partridge, A. C. Snoeren, T. Strayer, B. Schwartz, M. Condell, and I. Castineyra, "FIRE: flexible intra-AS routing environment," in *Proc. ACM SIGCOMM*, 2000.

THE USENIX ASSOCIATION

Since 1975, the USENIX Association has brought together the community of system administrators, developers, programmers, and engineers working on the cutting edge of the computing world. USENIX conferences have become the essential meeting grounds for the presentation and discussion of the most advanced information on new developments in all aspects of advanced computing systems. USENIX and its members are dedicated to:

- problem-solving with a practical bias
- fostering technical excellence and innovation
- encouraging computing outreach in the community at large
- providing a neutral forum for the discussion of critical issues

Membership Benefits

- Free subscription to *;login:*, the Association's magazine, both in print and online
- Online access to all Conference Proceedings from 1993 to the present
- Access to the USENIX Jobs Board: Perfect for those who are looking for work or are looking to hire from the talented pool of USENIX members
- The right to vote in USENIX Association elections
- Discounts on technical sessions registration fees for all USENIX-sponsored and co-sponsored events
- Discounts on purchasing printed Proceedings, CD-ROMs, and other Association publications
- Discounts on industry-related publications: see http://www.usenix.org/membership/specialdisc.html

For more information about membership, conferences, or publications, see http://www.usenix.org.

SAGE, a USENIX Special Interest Group

SAGE is a Special Interest Group of the USENIX Association. Its goal is to serve the system administration community by:

- Establishing standards of professional excellence and recognizing those who attain them
- Promoting activities that advance the state of the art or the community
- Providing tools, information, and services to assist system administrators and their organizations
- Offering conferences and training to enhance the technical and managerial capabilities of members of the profession

Find out more about SAGE at http://www.sage.org.

Thanks to USENIX & SAGE Supporting Members

Ajava Systems, Inc.
Cambridge Computer Services, Inc.
cPacket Networks
DigiCert® SSL Certification
EAGLE Software, Inc.
FOTO SEARCH Stock Footage and Stock Photography
Google
GroundWork Open Source Solutions
Hewlett-Packard
IBM
Infosys
Intel
Interhack
MSB Associates
NetApp
Oracle
Raytheon
Ripe NCC
rTIN Aps
Sendmail, Inc.
Splunk
Sun Microsystems, Inc.
Taos
Tellme Networks
UUNET Technologies, Inc.
VMware

ISBN-13: 978-1931971515
ISBN-10: 193197151X
90000
9 781931 971515